Optimizing distributed XML queries through localization and pruning

Patrick Kling
pkling@cs.uwaterloo.ca

M. Tamer Özsu
tozsu@cs.uwaterloo.ca

Khuzaima Daudjee
kdaudjee@cs.uwaterloo.ca

University of Waterloo
David R. Cheriton School of Computer Science
Waterloo, Canada

Technical Report CS-2009-13
March 2009

ABSTRACT

Distributing data collections by fragmenting them is an effective way of improving the scalability of relational database systems. The unique characteristics of XML data present challenges that require different distribution techniques to achieve scalability. In this paper, 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 individual 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.

1. INTRODUCTION

XML is commonly used to store data and to exchange them between a variety of systems. While centralized querying of XML data is increasingly well understood, the same is not true when the data are spread across multiple sites in a distributed system.

There are two main reasons why it is important to be able to query distributed XML data. First, the increasing size of XML data collections and the increasingly heavy workloads evaluated over these collections make it infeasible to scale centralized solutions. Distributing data collections and query processing workloads across multiple sites is therefore necessary in order to maintain good performance.

The second reason why querying distributed XML is an important problem lies in the fact that XML is frequently used to exchange data among a wide variety of systems. Therefore, XML data collections are often federated from multiple, independently managed sub-collections that originate at different sites. A distributed query evaluation strategy is well-suited to accessing these types of collections without having to ship large volumes of irrelevant data across the network and without having to maintain multiple copies of the same data at multiple sites solely for querying purposes.

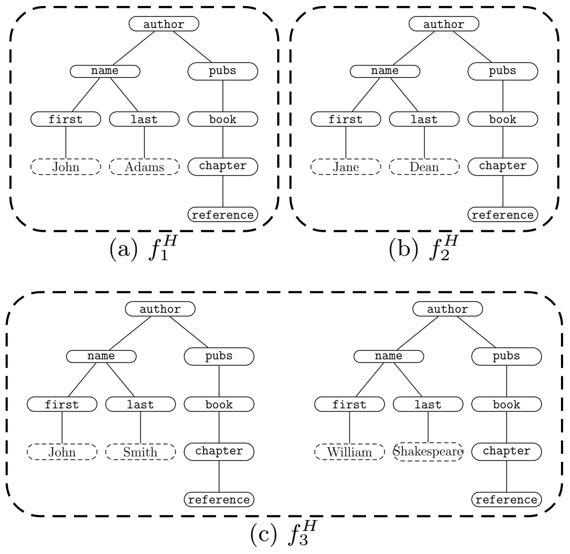
These two cases are analogous to distributed database systems [19] and data integration systems [15], respectively. In both cases, distributed query processing and optimization have been shown to be important, but the particular challenges differ significantly. In this paper, we focus on distributed XML query processing of the first type, namely a scenario where a database instance is distributed across multiple sites in order to improve performance.

Our distribution model for XML data supports horizontal fragmentation, which fragments data based on a selection defined by predicates, and vertical fragmentation, which fragments data by projecting to subsets of the types defined in the schema. Our definitions of horizontal and vertical fragmentation are based on the same semantics as those that are commonly used for relational data [19]. The characteristics of XML, such as its nested data model and structure-based queries, however, lead to a unique range of challenges and optimization opportunities for distributed querying that differ significantly from those seen in the relational scenario.

While a comprehensive study of the problem of querying XML in a distributed fashion is the subject of our broader work, this paper focuses on the issues of localization and pruning. We propose a localization technique that can transform a fragmentation-unaware query into sub-queries that can be evaluated in parallel at the individual sites in the system. We then present a novel technique that can prune sites for which it can infer that they do not contain data relevant for answering the query.

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 denoted as "P i" and "RP i" for now, we can see that `author` 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.



(a) f_1^H (b) f_2^H (c) f_3^H

Figure 1: A horizontally fragmented collection

Consider evaluating the following query (XQ1):

```
/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 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.

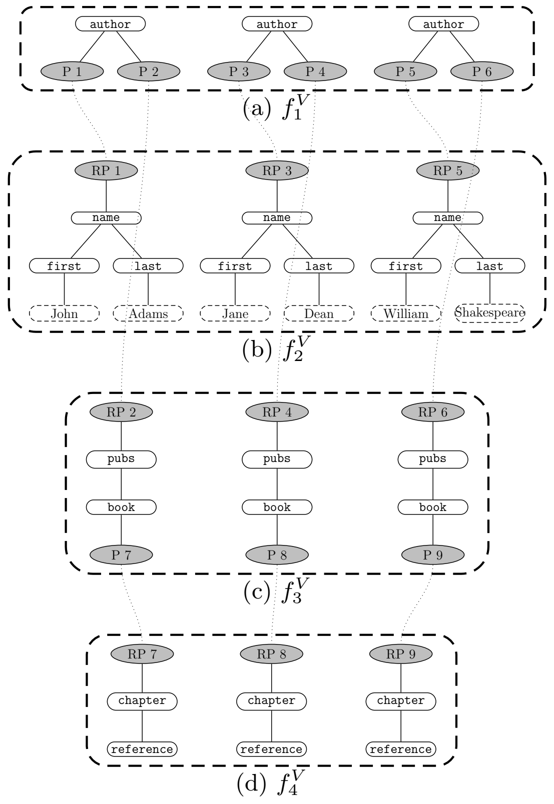
If we evaluate XQ1 on the vertically fragmented collection, in the general case, we have to access all four fragments. Fragment f_2^V is needed to evaluate the value constraint predicates, fragment f_4^V is needed to obtain result nodes and fragments f_1^V and f_3^V are needed to evaluate structural constraints. We will later present techniques that can avoid accessing some of these fragments.

To put our work in proper context, we note that the following issues need to be systematically addressed to develop a coherent set of techniques for distributed XML processing:

1. Fragmentation algorithm. The issue here is to develop algorithms that start with a non-distributed XML database and produce a distributed version. In this context, it is important to identify what "fragmentation" means in the context of XML data, what types of fragmentation are meaningful and how fragmentation schemas can be defined.

2. Data localization. This is the process whereby a query that is posed on a non-distributed version of the XML database is converted into a set of queries that are executed at the sites that may contain answers to the query.

3. Distributed query optimization. Although localization allows some optimization by focusing query execution at individual sites, there are global optimization opportunities and challenges for operations across multiple sites that go beyond this (e.g., distributed join, aggregation).



(a) f_1^V (b) f_2^V (c) f_3^V (d) f_4^V

Figure 2: A vertically fragmented collection

Our broader work addresses all of these issues, but in this paper we focus primarily on localization and pruning. 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. 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:

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 fragmentation-unaware query plans into equivalent distributed query plans.

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 have implemented these techniques within a real-life distributed XML database system, which has allowed us to obtain realistic experimental results.

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

Figure 3: A schema

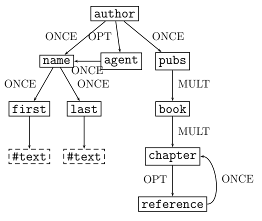


Figure 4: An XML schema graph

The remainder of this paper is structured as follows: Section 2 gives details on the data and query models that we employ. Section 3 describes how fragmentation layouts can be specified and how workload information can be used to derive a good specification. Our approach to query localization and fragment pruning is described in Sections 4 and 5, which deal with horizontal and vertical fragmentation, respectively. In Section 6, we evaluate the performance of this approach. In Section 7, we discuss related work. Finally, we present our conclusions in Section 8.

2. BACKGROUND

2.1 Data model

An XML collection can be described as a set of labeled, ordered trees. The structure of these trees is usually constrained by a schema, which specifies how elements may be nested and what the domain of their textual content is. We use a simple directed graph representation to describe this schema information 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 *items*. It is straightforward to translate a DTD or XML Schema into the graph representation, which we define next.

Definition 1. An XML *schema graph* is defined as a 5-tuple $(\Sigma, \Psi, s, d, \rho)$ where Σ is an alphabet of item types, ρ is the root item type, $\Psi \subseteq \Sigma \times \Sigma$ is a set of edges between item 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 an item of type σ_1 may contain an item of type σ_2. $s(\psi)$ denotes the cardinality of the containment represented by this edge: If $s(\psi) = \text{ONCE}$, then an item of type σ_1 must contain exactly one item of σ_2. If $s(\psi) = \text{OPT}$, then an item of type σ_1 may or may not contain an item of type σ_2. If $s(\psi) = \text{MULT}$, then an item of type σ_1 may multiple items of type σ_2. $m(\sigma)$ denotes the domain of the text content of an item of type σ, represented as the set of all strings that may occur inside such an item. Note that the definition of $m(\sigma)$ may include both the direct content of an item of type σ as well as the content of item types nested in σ. Figure 4 illustrates how the DTD shown in Figure 3 can be represented as a graph.

2.2 Query model and tree patterns

We focus on a query model that contains the following subset of XPath:

$$Q := \sigma \mid * \mid Q/Q \mid Q//Q \mid Q[Q] \mid Q[P]$$
$$P := \text{text}()\ o_\text{t}\ C_\text{text} \mid \text{val}()\ o_\text{n}\ C_\text{num} \mid P \wedge P \mid P \vee P \mid C_\text{pos}$$

Q defines arbitrarily nested path expressions with `child` and `descendant` axes and node tests for either a valid item type $\sigma \in \Sigma$ or a wildcard $*$ that matches all item types; P defines arbitrarily nested conjunctive and disjunctive constraints on the text or numerical content of an item or on its context position [9]; text() and val() represent the text or numerical value of a node; o_t and o_n represent arbitrary comparison operators that operate on text (such as = and starts-with()) or numerical values (such as < and $\leq$), respectively; C_text and C_num represent arbitrary text or numerical constants; C_pos denotes a numerical constant representing a context position. The class of queries that can be represented by this query model is sufficient to express a wide variety of realistic XPath queries [13]. Queries comprised of the primitives mentioned above can be expressed as tree patterns [6, 23], which we formalize as follows:

Definition 2. Let $(\Sigma, \Psi, s, m, \rho)$ be the schema of the data collection. A *tree pattern* is a 7-tuple $(N, E, r, \nu, \epsilon, T, c)$ where N is a set of pattern nodes, $E \subseteq N \times N$ is a set of pattern edges and (N, E, r) is a tree rooted at $r \in N$. For each $n \in N$, $\nu(n) \in \Sigma \cup \{*\}$ denotes a node test. For each $e \in E$, $\epsilon(e) \in \{\texttt{child}, \texttt{descendant}\}$ 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 items of type $\nu(n)$.

In addition to the class of XPath queries defined above, these tree patterns can be used to represent queries with multiple extraction points, i.e., queries whose results are comprised of tuples that consist of multiple document items. Therefore, the query model presented here allows for queries with complex results, which permits us to support a subset of XQuery that goes beyond simple path expressions.

In the following, we will refer to the tree pattern representation of a query as a *query tree pattern* (QTP). Query XQ1 given earlier can be expressed as the QTP depicted in Figure 5, where the `reference` node is an extraction point.

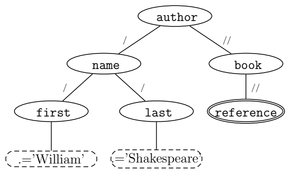


Figure 5: A query tree pattern (QTP)

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

3. FRAGMENTATION

3.1 Horizontal fragmentation

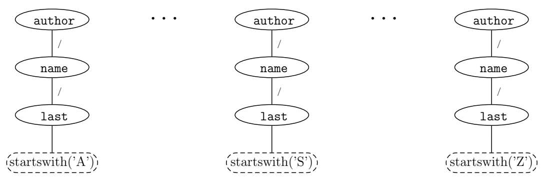


Figure 6: Set of fragmentation tree patterns (FTPs)

The horizontal fragmentation model 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 example include MathML [8] and CML [18].

A horizontal fragmentation is defined by a set of fragmentation predicates. Each fragment consists of the document trees that match the corresponding predicate. In order 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 $(\Sigma, \Psi, s, m, \rho)$. 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. Alternatively, instead of defining predicates as mutually exclusive, they could be arranged in a sequence and each document tree associated with the fragment that corresponds to the first matching predicate.

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

3.2 Vertical fragmentation

Our model of vertical fragmentation can handle collections that consist of a single or of 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 is defined by fragmenting the schema graph of the data collection into disjoint subgraphs. Formally, we define this using a vertical fragmentation function that assigns to each item type the fragment to which it belongs.

Definition 4. Let $(\Sigma, \Psi, s, m, \rho)$ be a schema graph. Then we can define a *vertical fragmentation function* $\phi : \Sigma \to F_\Sigma$ where F_Σ is a partitioning of Σ.

Figure 7 shows a fragmented schema graph that corresponds to the schema from Figure 4. The item types have been fragmented into four disjoint subgraphs. Fragment f_1^V consists of the item types `author` and `agent`, fragment f_2^V consists of the item types `name`, `first` and `last` along with their text content, fragment f_3^V consists of `pubs` and `book` and fragment f_4^V includes `chapter` and `reference`.

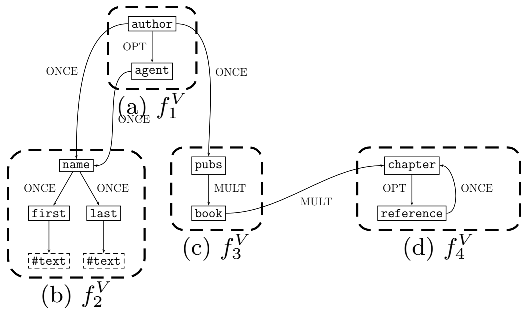


Figure 7: A vertically fragmented schema graph

In contrast to horizontally fragmented XML collections, where the data contained in each fragment is simply a subset of the document trees in the entire collection, vertically fragmented collections contain document tree edges that cross fragment boundaries. We represent such an edge as a pair of proxy nodes with matching IDs: a *proxy* node in one fragment corresponding to a *root proxy* in another fragment. The collection shown in Figure 2 has been fragmented according to the fragmented schema from Figure 7. We denote the proxy pair with ID i as P i and RP i. The proxy node P 1 in fragment f_1^V thus corresponds to the root proxy node RP 1 in fragment f_2^V.

Vertical fragments generally consist of multiple unconnected pieces of XML data. To distinguish them from the entire document trees stored in horizontal fragments, we refer to them as *document snippets*. In Figure 2, for example, fragment f_1^V consists of three snippets each of which consists of the author item of one of the documents in the collection.

3.3 Designing fragmentation layouts

Using the mechanisms for specifying horizontal and vertical fragmentation, a fragmentation algorithm can create a fragmentation layout that is suitable for a given workload. It also needs to take into account the specific performance goals, such as throughput or response times. While a detailed study of fragmentation algorithms is beyond the scope of this paper, we briefly point out some of the considerations involved in designing a fragmentation layout.

For horizontal fragmentation, spreading the document trees relevant for a given query across a large number of fragments generally improves response time by evaluating that query in parallel at multiple sites. Concentrating the data needed for each common query in a small set of fragments can improve throughput by reducing inter-query interference.

The number of fragments that are accessed by a given query plan can be tuned by choosing FTPs based on the value constraints encountered in the QTP representation of the query. Based on knowledge of the distributed query evaluation technique used, the cost for multiple candidate fragmentation designs can be estimated and a suitable design chosen.

For a vertical fragmentation layout, on the other hand, it is generally preferable to minimize the number of fragments needed to answer a given query. This reduces the portion of the collection that needs to be inspected and at the same time limits the number of joins that have to be performed to combine results from multiple fragments, which improves both response time and throughput.

Another important consideration when designing a fragmentation layout lies in the trade-off between processing cost on the one hand and communication cost on the other. This is particularly important with vertical fragmentation, where results from multiple fragments have to be joined together.

4. LOCALIZATION AND PRUNING WITH 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 [19], 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 $f \in F$, 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 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

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

is a naïve 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 [11]. 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 items in the same document. This may be a reasonable trade-off in many use cases.

4.1 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, however, accesses every fragment in the collection, which can lead to poor performance.

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 data and can be encoded in a compact manner. This makes it feasible to replicate them at all sites as metadata.

In relational systems, fragments are usually pruned based on an algebraic representation of a distributed query [19]. Here, however, the QTP presents a simpler abstraction that contains all the information necessary to make pruning decisions. We therefore prune based on the QTP representation 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 for which we can show that they cannot be satisfied at the same time as the QTP.

To solve this problem, we transform QTP and FTPs into a canonical representation. We then traverse them 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.

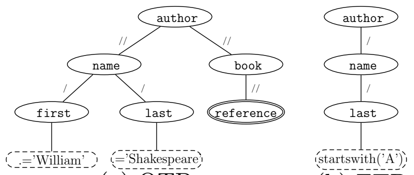
4.2 Transformation to canonical form

The goal of transforming tree patterns into a canonical form is to make sure each pattern node refers to a unique item within the context of a single document tree. In general, pattern nodes may match more than one item in a given document tree. A constraint associated with such a pattern node is satisfied if one of the matching items 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 items in the same document.

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

Item types reached via MULT edges Item 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` items corresponding to a single `pubs` item.

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` item that is the direct child of a given `author` item or by a `name` item that is reachable through an intermediate `agent` item. Because of this, even though the constraints on the author's last name imposed by the FTP shown in Figure



(a) QTP (b) FTP

Figure 8: QTP and FTP that are not contradictory

8(b) and the QTP shown in Figure 8(a) seem to cause these two patterns to be contradictory, they are actually 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 an item type may contain multiple other item types.

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

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

In order to convert a tree pattern into a canonical tree pattern, all disallowed primitives have to either be removed or converted into an equivalent canonical form. It is important to note that canonical tree patterns are strictly less expressive than arbitrary tree patterns. Therefore, when a tree pattern is transformed to a canonical tree pattern, the result is not generally equivalent to the original tree pattern. Instead, the canonical tree pattern matches a superset of the document trees that match the original tree pattern. Since canonical 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 as possible of the information present in the original pattern so that this information can be exploited for pruning decisions.

In the following, we describe our algorithm for transforming tree patterns into the canonical form.

4.2.1 Unrolling descendant steps

The unrolling of `descendant` steps can be succinctly implemented as a manipulation of the directed graph representation of the schema. 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 (Algorithm 1, lines 31 and 32) . This yields a graph that contains all the intermediate item 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 is a cycle in this schema subgraph, we discard the `descendant` step and all the pattern nodes that occur below it (Algorithm 1, line 33). This is necessary because the presence of a cycle implies that a matching item 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

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 nodes whose children have
           yet to be checked
variable : $N''$ // set of nodes to be inserted
variable : $E''$ // set of 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 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 node to the canonical 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 $\nu(x)$,
               $\nu(y)$ reachable from $\sigma$ in $(\Sigma, \Psi)\}$
32             $\Psi' \leftarrow \{(\sigma_1, \sigma_2) \in \Psi \mid \sigma_1, \sigma_2 \in \Sigma'\}$
33             if $(\Sigma', \Psi')$ is acyclic and $\nexists \psi \in \Psi'$ with
               $s(\psi) = MULT$ then
34                 $\nu'(y) \leftarrow \nu(y)$
35                 $(N'', E'') \leftarrow$
                   unrolldesc$(q', y', \Sigma', \Psi', \nu(x))$
36                 $N' \leftarrow N' \cup N'' \cup \{y'\}$
37                 $E' \leftarrow E' \cup E''$
38                 $Q \leftarrow Q \cup \{(y, y')\}$
39 $\forall e' \in E', \epsilon'(e') \leftarrow$ child
40 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  : nodes $N''$, 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 ⊕ 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$(N'', E'')$
```

example, that we want to unroll the step `book//reference`. We can observe that there is a cycle involving the item 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.

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). In cases where a schema node has more than one child, an intermediate choice node is inserted (Algorithm 2, lines 6–13, denoted by ⊕), which signifies that the subsequent branch of the pattern can be satisfied by a match for any of the child nodes.

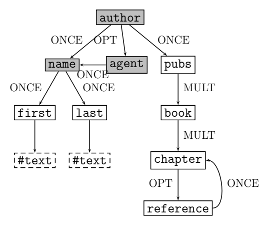


Figure 9: Schema restricted to nodes reachable from `author` and from which `name` is reachable

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 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.

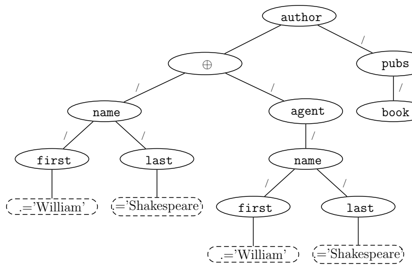


Figure 10: Pattern after unrolling descendant steps

4.2.2 Removing wildcard nodes

We convert wildcard nodes whenever they unambiguously refer to a specific item type (Algorithm 1, lines 18–21). For example, by relying on the schema shown in Figure 4, the step `agent/*` can be translated to `agent/name`. It is also possible to convert wildcard nodes that can refer to more than one item type by introducing choice nodes into the pattern in a procedure that is largely analogous to the way `descendant` steps are unrolled.

4.2.3 Removing nodes referring to items with multiple occurrences in the same context

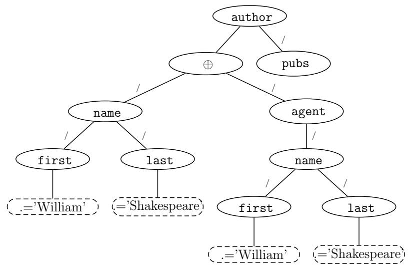


Figure 11: Canonical pattern

In general, a meaningful conversion of pattern nodes corresponding to items 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 item (e.g. `pubs/book[1]`). In this case, we

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 ← false // constraint violation found
3   else if $\nu(r) = \oplus$ then
4       // check if at least one choice leads to satisfiable constraints
5       result ← 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 ← result ∨ traverse($(N, E, n), (N', E', y)$)
9           else
10              result ← true
11  else if $\nu'(r') = \oplus$ then
12      // check if at least one choice leads to satisfiable constraints
13      result ← 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 ← result ∨ traverse($(N, E, y), (N', E', n')$)
17          else
18              result ← true
19  else
20      // check all child nodes
21      result ← 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 ← result ∧ traverse($(N, E, n), (N', E', y)$)
25  return result
```

could retain the pattern node and exploit its associated constraints for pruning (not shown in Algorithm 1). In the example from Figure 10, we need to remove the `book` node since the schema indicates that a `pubs` item may have multiple children of type `book`. The resulting canonical pattern is shown in Figure 11.

4.3 Traversal and pruning

After transforming QTP and FTP into canonical tree patterns, Algorithm 3 traverses both patterns simultaneously. Only nodes occurring in both patterns are visited. For each pair of corresponding nodes, we check whether the value constraints in one pattern contradict those in the other pattern. Since in canonical tree patterns each node corresponds to a unique item 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.

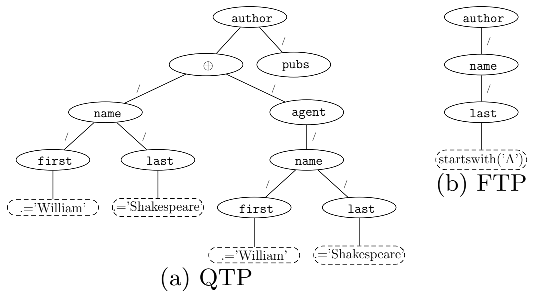


Figure 12: Canonical QTP and FTP that are not contradictory

Special care has to be taken when a choice node is encountered (Algorithm 3, lines 3–18). 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.

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 item 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 equivalence 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.

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.

4.4 Analysis and optimization

While it may seem that the transformation and traversal of QTP and FTPs could pose a significant overhead, there

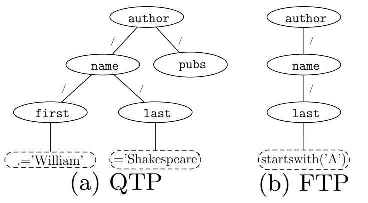


Figure 13: Canonical QTP and FTP that are contradictory

are a number of consideration that reduce this impact. The transformation of the FTPs only has to be performed once when the fragmentation is set up. It does therefore 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 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 [22], 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. Because this 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 $O((|E^{\text{QTP}}_{\texttt{desc}}|\ |\Sigma|\ |E^{\text{QTP}}_{\texttt{child}}|+|E^{\text{QTP}}_{\texttt{desc}}|^2\ |\Sigma|^2)\ (|E^{\text{FTP}}_{\texttt{desc}}|\ |\Sigma|\ |E^{\text{FTP}}_{\texttt{child}}|+|E^{\text{FTP}}_{\texttt{desc}}|^2\ |\Sigma|^2)\ |F|)$.

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 canonical 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.

5. LOCALIZATION AND PRUNING WITH VERTICAL FRAGMENTATION

Based on the definition of vertical fragmentation, we define a strategy for evaluating QTPs on a vertically fragmented collection. First, we decompose the global QTP into a set of local QTPs corresponding to 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). Finally, we combine the results of this process by joining the matches derived from individual fragments based on their proxy/root proxy IDs.

The decomposition of a global QTP into a set of local QTPs directly follows the schema graph. After unrolling wildcard nodes using the same procedure employed by horizontal pruning, Algorithm 4 divides the global QTP into a set of subtrees each of which consist of pattern nodes that match items 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 a subtree a to one in a subtree b is converted to a pattern node matching a proxy in a and a pattern node matching a root proxy in b. These 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, they are unrolled into child steps according to the same procedure that is used by the horizontal transformation algorithm (not shown in Algorithm 4). 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 a 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, namely `author/pubs` and `book/chapter`. Therefore, an additional local QTP corresponding to fragment f^V_3 (which contains the `pubs` and `book` item types) is introduced, even if there is

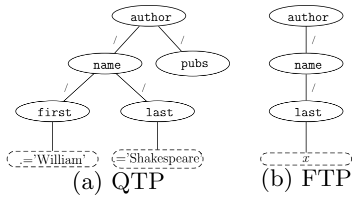


Figure 14: Canonical QTP and abstract FTP

Algorithm 4: Vertical localization

```
input  : global QTP $(N, E, r, \nu, \epsilon, T, c)$, schema
         $(\Sigma, \Psi, s, m, \rho)$, vertical fragmentation
         function $\phi : \Sigma \to F_\Sigma$
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     $r_2 \leftarrow rp_i$
```

no pattern node in the global QTP that refers to item types in this fragment.

Localizing the global QTP shown in Figure 5 yields the set of local QTPs shown in Figure 15(a)–(d). Note that while the unrolling of cross-fragment descendant steps can increase the size of the QTPs, this is mitigated by the pruning technique that we propose. In addition, local optimization techniques can evaluate linear paths of child steps efficiently by taking schema information into account.

If the global QTP does not reach a certain 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.

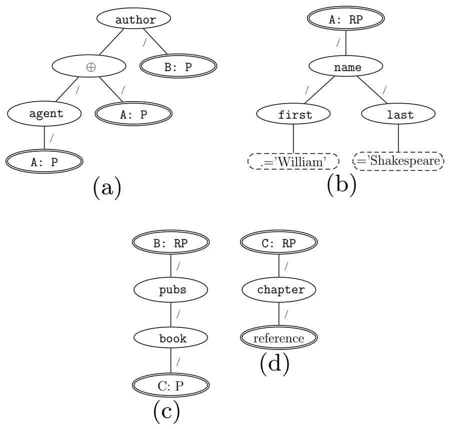


Figure 15: Local QTPs

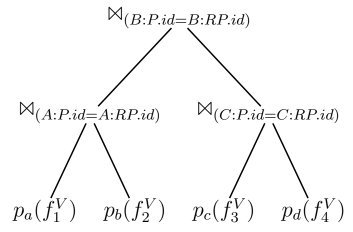

$\bowtie_{(B:P.id=B:RP.id)}$
$\bowtie_{(A:P.id=A:RP.id)}$
$\bowtie_{(C:P.id=C:RP.id)}$
$p_a(f_1^V)$
$p_b(f_2^V)$
$p_c(f_3^V)$
$p_d(f_4^V)$

Figure 16: Initial vertical plan

Local QTPs are then evaluated on the corresponding fragments using existing, local tree pattern evaluation techniques. Finally, whenever two subtrees of the global QTP share a cross-fragment edge, the plans corresponding to their local QTP representations are joined together based on the IDs of the proxy pair representing this edge. Assuming that p_a, p_b, p_c and p_d represent local plans that evaluate the QTPs shown in Figures 15(a), (b), (c) and (d), respectively, we could evaluate the query using the plan shown in Figure 16.

5.1 Skipping fragments

The localization strategy for vertical fragmentation avoids accessing fragments that are not reached by the global QTP. Intermediate fragments, however, have to be accessed even if no constraints are evaluated on them. In our example, we have to evaluate the local QTP shown in Figure 15(c) and therefore access fragment f_3^V in order to determine, for example, that proxy P 2 in fragment f_1^V is indirectly connected to root proxy RP 7 in fragment f_4^V. We propose a pruning technique that allows us to avoid accessing such intermediate fragments.

So far, we have made no assumptions on how IDs are assigned to proxy pairs when data are inserted into a vertically distributed collection. We now propose a proxy numbering scheme called *skipping IDs* that is based on the Dewey numbering system[1] and that allows us to infer whether a root proxy node is the descendant of a given proxy node. To define this numbering scheme, we need to distinguish between the following two cases: **(i)** If a document snippet does not have a root proxy node as its root (i.e., if the snippet contains the root element of a document tree in the collection), then the proxy nodes in this fragment (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, fragment f_1^V is the only such fragment. The proxy nodes in this fragment therefore receive numeric IDs, which means that P 1 through P 6 are already numbered in accordance with our numbering scheme. **(ii)** If a document snippet is rooted at a root proxy node, then the ID of each of their proxy nodes is prefixed by the ID of the root proxy node of the snippet, followed by a numeric identifier that is unique within this snippet. In Figure 2, fragments f_2^V, f_3^V and f_4^V consist of snippets that are rooted at a root proxy. However, only fragment f_3^V contains proxy nodes. Therefore, only P 7 through P 9 have to be renumbered. P 7 is part of a snippet that is rooted at the root proxy node RP 2. We would therefore have to renumber it to P

[1]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 also applicable to these representations.

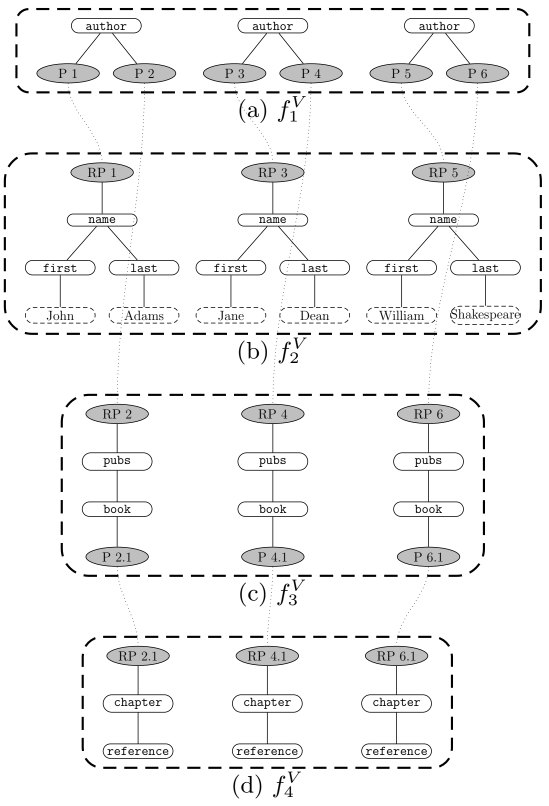


Figure 17: A vertically fragmented collection with skipping IDs

2.1. Similarly, P 8 would be renumbered to P 4.1 and P 9 to P 6.1. Figure 17 shows the result of renumbering the proxy nodes in the vertically fragmented collection shown in Figure 2.

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 16 to the simpler plan shown in Figure 18, which avoids accessing fragment f_3^V.

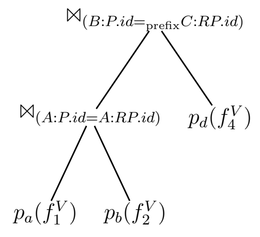

$\bowtie_{(B:P.id =_{\text{prefix}} C:RP.id)}$
$\bowtie_{(A:P.id = A:RP.id)}$
$p_d(f_4^V)$
$p_a(f_1^V)$
$p_b(f_2^V)$

Figure 18: Skipping vertical plan

5.2 Dealing with structural ambiguity

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 if they are not needed for evaluating value constraints. To illustrate this, consider the modified fragmented schema shown in Figure 19, which adds `article` as an additional type of publication. 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 snippets in fragment $f_4^{V'}$ that are part of a book and those that are part of an article.

We propose a technique that lets us skip such fragments by keeping track of some structural information in the proxy IDs if there is ambiguity. We define structural ambiguity as follows: Let f_a be a fragment whose snippets are rooted at root proxy nodes and that snippets 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 snippet 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 fragmented schema shown in Figure 19, there is one instance of structural ambiguity. The 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.

Label paths contain sufficient information to evaluate structural constraints such as those seen in Figure 15(c). In com-

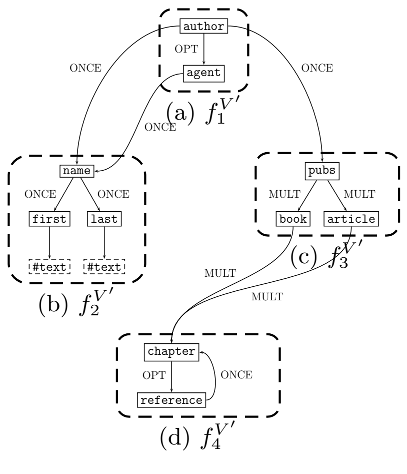


Figure 19: A modified fragmented schema graph

Q1	`/open_auction[./interval/end[. = xs:date('12/28/2001')]][initial > 200]//item/name`
Q2	`/open_auction[./interval/end[. >= xs:date('01/01/1998')][. < xs:date('12/28/1998')]][initial > 200]//item/name`
Q3	`/open_auction[./interval/end[. >= xs:date('01/01/1998')][. < xs:date('12/28/1999')]][initial > 200]//item/name`
Q4	`/open_auction[./interval/end[. >= xs:date('01/01/1998')][. < xs:date('12/28/2000')]][initial > 200]//item/name`
Q5	`/open_auction[./interval/end[. >= xs:date('01/01/1998')][. < xs:date('12/28/2001')]][initial > 120]//item/name`
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']`

Table 1: Queries used in experiments

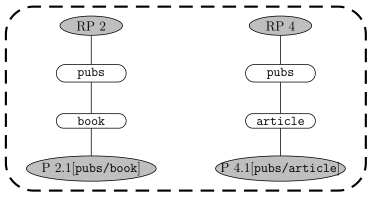
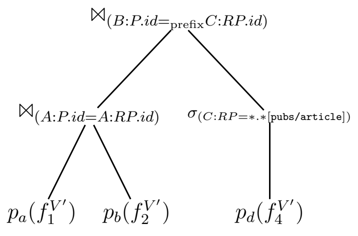


Figure 20: Fragment f_3^V with label path IDs

bination with skipping IDs, they therefore allow us to evaluate the query using the plan shown in Figure 21, which avoids accessing $f_3^{V'}$.



$\bowtie_{(B:P.id=_{\text{prefix}}C:RP.id)}$
$\bowtie_{(A:P.id=A:RP.id)}$
$\sigma_{(C:RP=*.*[\text{pubs/article}])}$
$p_a(f_1^{V'})$
$p_b(f_2^{V'})$
$p_d(f_4^{V'})$

Figure 21: Label path plan

5.3 Analysis

Both skipping IDs and label paths are inserted at fragmentation time and whenever data is 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 constants used in value constraints. This allows us to perform pruning at query compile time, thereby minimizing the run-time overhead introduced by pruning.

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

6. PERFORMANCE EVALUATION

We implemented our techniques within the native XML database system NATIX [4], to validate the approach and to perform realistic experiments. Our goal is to show that our techniques can improve the performance of query processing through distribution and pruning.

The data collection we use for our experiments is based on the XMark benchmark [21]. XMark provides a single document consisting of on-line auction data that can be scaled to a wide variety of sizes. Since the horizontal fragmentation model assumes a collection that consists of multiple separate document trees rather than one large document tree, we denormalize the XMark data into multiple documents by duplicating references shared between more than one auction. As shown in Table 2, we use collections that are 35 MB, 350 MB and 3.5 GB in size after denormalization.

All experiments are conducted on virtualized Linux instances within Amazon's Elastic Compute Cloud. For each fragment, we use a separate "small" instance, which provides 1.7 GB of memory, a single-core 32 bit CPU and 160 GB of storage. In addition to this, we use a "medium" dual-core instance to dispatch the queries. All instances are running in the same "availability zone", providing low-latency, high-throughput communication.

6.1 Horizontal fragmentation

For the horizontal fragmentation model, the primary goal of our evaluation is to show that our pruning algorithm significantly increases throughput in a distributed system, without adding overhead that would significantly reduce response times.

6.1.1 *Balanced fragmentation*

The data generated by XMark are uniformly distributed across the years 1998-2001 with respect to the end date of the auction. We exploit this property to define a balanced horizontal fragmentation consisting of 16 fragments of equal size, each of which corresponds to the auctions ending in one particular quarter of one of the four years. Figure 22 shows the FTP corresponding to one of the fragments.

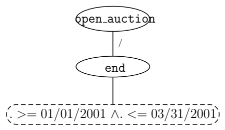


Figure 22: XMark FTP

We use the queries Q1 through Q5 shown in Table 1. Q1 is focused on one particular date. Our pruning algorithm

Scale fac.	XMark size	Conv. size	Number of docs.
0.1	12 MB	38 MB	1200
1	112 MB	338 MB	12000
10	1.1 GB	3.5 GB	120000

Table 2: XMark instances used

leverages this to produce a plan that only needs to inspect a single fragment. Q2-Q5 require 4, 8, 12, and all 16 fragments, respectively.

To measure the throughput achieved by the various techniques, we use 24 processes on the dispatcher instance that are constantly issuing queries on the distributed system. While this may seem like a large number of processes for a dual-core machine, the dispatcher spends most of its time waiting for results to arrive from other sites, where the bulk of query processing is performed. For each query instance, the dispatcher randomly varies the dates mentioned in the query while retaining the length of the date range as specified in Q1-Q5. This results in a query workload whose results are evenly distributed across the four years.

The results of this experiment are shown in Table 3. For each collection size and query, we see, from left to right in the next three columns, the throughput rates achieved by a single, centralized NATIX instance, a distributed system without pruning, and a distributed system using the pruning techniques presented in this paper. Even without pruning, distribution significantly increases throughput. Enabling pruning further improves throughput by a factor of up to 12. Naturally, the impact of pruning is most pronounced for Q1 where a single date is selected and where our pruning algorithm can therefore avoid accessing all but one of the 16 fragments. 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.

This illustrates the importance of a fragmentation layout that is well suited to the workload. It shows that 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 such cases, however, distributed evaluation by far outperforms centralized querying.

		Throughput (queries/sec.)				
			Balanced frag.		Skewed frag.	
Col. size	Query	Central	Dist.	Pruning	Dist.	Pruning
35 MB	Q1	1.64	7.13	28.08	4.92	19.93
	Q2	1.44	6.94	16.64	4.67	11.75
	Q3	1.34	6.39	6.65	4.53	5.56
	Q4	1.24	6.32	6.34	4.24	4.53
	Q5	1.17	6.40	6.40	3.88	3.86
350 MB	Q1	0.20	2.26	24.55	0.80	2.97
	Q2	0.18	2.14	4.50	0.71	2.21
	Q3	0.17	1.72	1.78	0.71	1.10
	Q4	0.15	1.74	1.76	0.62	0.67
	Q5	0.13	1.62	1.64	0.56	0.54
3.5 GB	Q1	< 0.01	0.30	3.60	0.08	0.33
	Q2	< 0.01	0.24	0.50	0.08	0.29
	Q3	< 0.01	0.24	0.24	0.08	0.13
	Q4	< 0.01	0.21	0.22	0.08	0.08
	Q5	< 0.01	0.18	0.19	0.06	0.07

Table 3: Throughput

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 fragmentation, which places the auctions ending in 1998 in a single fragment, splits the auction ending in 1999 into two fragments, those ending in 2000 into 4 fragments and the auctions of 2001 into the remaining 9 fragments.

The results are shown on the rightmost two columns of Table 3. Even in the presence of skew, distribution results in a significant boost in performance over centralized querying in all cases. Adding pruning leads to a further improvement wherever the queries are sufficiently focused with respect to the end dates. Just as in the balanced scenario, this effect is most pronounced for Q1, whose plan can be pruned to a single fragment.

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

6.1.3 *Response time*

Next, we verify that the improvement in throughput does not come at the expense of increased response times. We measure the time it takes to evaluate the queries Q1-Q5 in an otherwise idle system using the balanced fragmentation. The results are shown in Table 4. Distribution has a large positive impact on response times when compared to centralized processing. Enabling pruning does not lead to increased response times. Instead, it appears to have a small positive impact for Q1 since the dispatcher only has to wait for a single site to finish, which is not necessarily the slowest.

		Response time (seconds)		
Col. size	Query	Central	Dist.	Pruning
35 MB	Q1	0.75	0.27	0.19
	Q2	0.78	0.24	0.22
	Q3	0.81	0.27	0.25
	Q4	0.94	0.25	0.27
	Q5	0.95	0.25	0.27
350 MB	Q1	4.97	0.57	0.49
	Q2	5.61	0.65	0.66
	Q3	6.05	0.66	0.66
	Q4	6.50	0.65	0.70
	Q5	6.85	0.67	0.67
3.5 GB	Q1	403.65	3.43	3.26
	Q2	407.03	4.63	4.75
	Q3	403.65	4.80	4.74
	Q4	409.91	4.74	4.68
	Q5	405.96	5.46	5.41

Table 4: Response time (horizontal)

6.1.4 *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. 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. The results for the largest collection are shown in Figure 23. The remaining results are given in Table 5, which shows, from left to right, the collection size, the query, the number of fragment accessed after pruning, the number of fragments that return part of the result and the percentage of fragments accessed that return part of the result. 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.

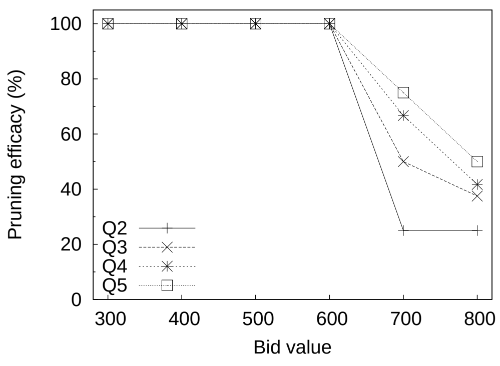


Figure 23: Pruning efficacy

6.2 Vertical fragmentation

The goal of the evaluation of our techniques for vertical fragmentation is to demonstrate that it can significantly improve performance in a distributed system. We also show how this increase in performance correlates with the effectiveness of our pruning algorithms.

6.2.1 *Response time*

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 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 denormalized XMark collections, which we fragment into six vertical fragments based on the fragmented schema shown in simplified form in Figure 24. This results in a skewed fragmentation because different item types in the collection occur with different frequencies. We evaluate the queries Q6-Q10 shown in Table 1. Q6 only involves a single fragment (shown in Figure 24(a)). Previous work has shown that this is the ideal case for vertical fragmentation [3]. The remaining queries, however, reach five of the six fragments in the collection (Figure 24(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 layout will therefore aim to reduce the frequency of this scenario, although we cannot always avoid it. One of the goals of this experiment is to show that our 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.

Table 6 shows, from left to right, the response times obtained by centralized querying, distributed querying without skipping, pruning based on skipping IDs and pruning based on skipping IDs as well as label paths. For Q6, which rep-

Col. size	Query	Bid	F. q'd	F. res.	Eff.
35 MB	Q2	200	4	4	100.0%
		300	4	4	100.0%
		400	4	1	25.0%
		500	4	1	25.0%
		600	4	1	25.0%
		700	4	0	0.0%
		800	4	0	0.0%
	Q3	200	8	8	100.0%
		300	8	8	100.0%
		400	8	3	37.5%
		500	8	3	37.5%
		600	8	3	37.5%
		700	8	0	0.0%
		800	8	0	0.0%
	Q4	200	12	12	100.0%
		300	12	12	100.0%
		400	12	6	50.0%
		500	12	5	41.7%
		600	12	5	41.7%
		700	12	1	8.3%
		800	12	0	0.0%
	Q5	200	16	16	100.0%
		300	16	16	100.0%
		400	16	10	62.5%
		500	16	7	43.8%
		600	16	6	37.5%
		700	16	2	12.5%
		800	16	1	6.3%
350 MB	Q2	200	4	4	100.0%
		300	4	4	100.0%
		400	4	4	100.0%
		500	4	3	75.0%
		600	4	2	50.0%
		700	4	1	25.0%
		800	4	1	25.0%
	Q3	200	8	8	100.0%
		300	8	8	100.0%
		400	8	8	100.0%
		500	8	7	87.5%
		600	8	4	50.0%
		700	8	3	37.5%
		800	8	3	37.5%
	Q4	200	12	12	100.0%
		300	12	12	100.0%
		400	12	12	100.0%
		500	12	11	91.7%
		600	12	7	58.3%
		700	12	6	50.0%
		800	12	5	41.7%
	Q5	200	16	16	100.0%
		300	16	16	100.0%
		400	16	16	100.0%
		500	16	15	94.8%
		600	16	10	62.5%
		700	16	8	50.0%
		800	16	7	43.8%
3.5 GB	Q2	200	4	4	100.0%
		300	4	4	100.0%
		400	4	4	100.0%
		500	4	4	100.0%
		600	4	4	100.0%
		700	4	1	25.0%
		800	4	1	25.0%
	Q3	200	8	8	100.0%
		300	8	8	100.0%
		400	8	8	100.0%
		500	8	8	100.0%
		600	8	8	100.0%
		700	8	4	50.0%
		800	8	3	37.5%
	Q4	200	12	12	100.0%
		300	12	12	100.0%
		400	12	12	100.0%
		500	12	12	100.0%
		600	12	12	100.0%
		700	12	8	66.7%
		800	12	5	41.7%
	Q5	200	16	16	100.0%
		300	16	16	100.0%
		400	16	16	100.0%
		500	16	16	100.0%
		600	16	16	100.0%
		700	16	12	75.0%
		800	16	8	50.0%

Table 5: Pruning efficacy

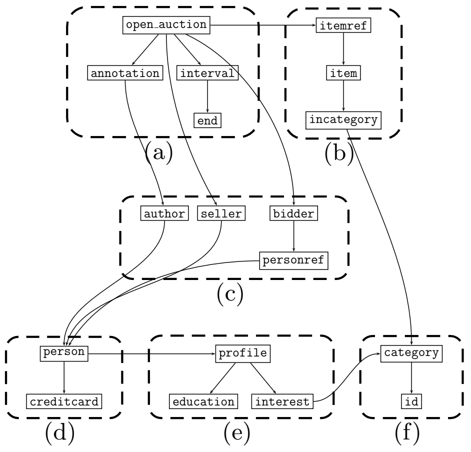


Figure 24: Vertically fragmented schema graph used for performance evaluation

		Response time (seconds)			
Col. size	Query	Central	Dist.	Skip.	Label
35 MB	Q6	0.48	0.25	0.23	0.28
	Q7	2.53	6.57	1.61	1.61
	Q8	1.95	6.40	3.37	1.62
	Q9	1.71	5.19	3.71	1.99
	Q10	1.39	4.87	4.62	2.98
350 MB	Q6	3.58	1.78	1.82	1.83
	Q7	22.09	77.72	14.90	14.90
	Q8	17.23	77.79	37.30	14.95
	Q9	14.57	64.50	46.30	23.30
	Q10	11.82	63.54	59.22	36.75
3.5 GB	Q6	374.77	17.67	17.41	17.60
	Q7	416.95	829.09	161.95	160.76
	Q8	403.22	826.21	398.13	160.84
	Q9	398.37	684.81	489.48	251.92
	Q10	388.86	672.18	622.85	387.09

Table 6: Response time (vertical)

resents the ideal case for vertical fragmentation, we can see that all three distributed techniques significantly outperform the centralized approach. This is because they only need to access a single fragment, rather than the entire collection.

For the remaining queries, simple distributed evaluation is much slower than centralized querying due to the large number of joins that need to be performed. By using our pruning techniques, however, we can achieve performance that surpasses that of a centralized system on the 3.5 GB collection. The benefit of pruning decreases as the number of constraints in the query increases. We can also see that the impact of pruning with label paths is generally much greater than that of pruning based skipping IDs alone. Only for Q6 and Q7 do label paths not offer any additional benefit.

6.2.2 *Pruning efficacy*

In order to examine why pruning techniques are so effective in improving distributed querying performance, we examine the number of fragments that the distributed plans access. As shown in Table 7, Q6 is answered by accessing a single fragment, regardless of whether we prune or not. This explains why all three distributed approaches yield comparable performance in this case. For Q7, both pruning techniques do equally well by generating plans that access only a single fragment. For Q8, Q9 and Q10, where label paths improve the performance of pruning, the label path-based pruning technique results in a plan that accesses fewer fragments than that produced by skipping ID-based pruning.

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

Table 7: Number of fragments accessed

6.3 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 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

Much of existing work on distributed XML query processing assumes a distribution model without an explicit fragmentation specification [1, 2, 7, 10]. While there are certain optimizations that can be performed in the absence of a fragmentation specification and while the flexibility of this approach is certainly appealing, having a well-specified fragmentation is a significant asset for effective distributed query optimization. When freely distributing data in order to improve query performance, such a specification can be obtained. One of the approaches that does not rely on a fragmentation specification and that supports a query model similar to ours is based on the idea of computing partial matches at each fragment and then combining them [10]. This provides impressive complexity properties although it is limited to queries with a single extraction point. Unlike the work reported here, however, there is no focus on eliminating irrelevant fragments from a distributed query plan.

Another area of active research has been query language extensions that include communication and distribution primitives [12, 20, 24]. These approaches cater primarily to a data integration scenario. They might, however, be useful as a backend language for a distributed database system.

In the area of specifying structure-based XML fragmentation, there are a number of approaches that emulate the horizontal and vertical fragmentation seen in relational distributed database systems [3, 5, 14, 16, 17]. Our notion of horizontal fragmentation is inspired by this work. One of these approaches [3] alludes to the possibility of pruning irrelevant horizontal fragments. However, the authors do not provide details on how this pruning could be performed.

It is possible to follows a different approach to specifying a vertical fragmentation collection [5]. Instead of basing a fragmentation purely on the item types present in the schema, this approach defines vertical fragmentation in terms of label paths from the root of the document. In the example shown in Figure 7, this would allow the separation of `name` elements that are reachable directly through an `author` element from those that are reachable via an `agent` element. While our definition does not directly support this, the same effect can be achieved by fragmenting vertically as shown in Figure 7 and subsequently performing a horizontal fragmentation of `name`. Following our definition, a vertical fragmentation of XML data is purely based on the schema. This enables us to use skipping optimizations, which would be more difficult to achieve in a path-based approach. Some existing techniques [5] make structural information available outside the corresponding fragment by using an approach that is similar to the label paths employed by our technique. In contrast to the work presented here, however, they store this information for all document nodes, whereas we only store it for proxy nodes and only if there is indeed ambiguity.

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 horizontal fragment pruning algorithm 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 designing a vertical fragmentation.

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 particular optimizations are possible for a hybrid of vertical and horizontal fragmentation.

9. REFERENCES

[1] 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.

[2] S. Abiteboul, A. Bonifati, G. Cobéna, I. Manolescu, and T. Milo. Dynamic XML documents with distribution and replication. In *Proc. of ACM SIGMOD*, 2003.

[3] A. Andrade, G. Ruberg, F. A. Baião, V. P. Braganholo, and M. Mattoso. Efficiently processing XML queries over fragmented repositories with PartiX. In *Current Trends in Database Technology, EDBT*, 2006.

[4] M. Brantner, S. Helmer, C.-C. Kanne, and G. Moerkotte. Full-fledged algebraic XPath processing in Natix. In *Proc. of ICDE*, 2005.

[5] J.-M. Bremer and M. Gertz. On distributing XML repositories. In *Proc. of WebDB*, 2003.

[6] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *Proc. of ACM SIGMOD*, 2002.

[7] P. Buneman, G. Cong, W. Fan, and A. Kementsietsidis. Using partial evaluation in distributed query evaluation. In *Proc. of VLDB*, 2006.

[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] J. Clark and S. DeRose. XML Path Language (XPath), 1999. http://www.w3.org/TR/xpath/.

[10] G. Cong, W. Fan, and A. Kementsietsidis. Distributed query evaluation with performance guarantees. In *Proc. of ACM SIGMOD*, 2007.

[11] 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/.

[12] M. F. Fernàndez, T. Jim, K. Morton, N. Onose, and J. Siméon. Highly distributed XQuery with DXQ. In *Proc. of ACM SIGMOD*, 2007.

[13] Z. G. Ives, A. Y. Halevy, and D. S. Weld. An XML query engine for network-bound data. *The VLDB Journal*, 11(4):380–402, 2002.

[14] 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.

[15] M. Lenzerini. Data integration: a theoretical perspective. In *Proc. of PODS*, 2002.

[16] H. Ma and K.-D. Schewe. Fragmentation of XML documents. In *Proc. of SBBD*, 2003.

[17] H. Ma and K.-D. Schewe. Heuristic horizontal XML fragmentation. In *Proc. of CAiSE*, 2005.

[18] P. Murray-Rust. Chemical markup language. *World Wide Web Journal*, 2(4):135–147, 1997.

[19] M. T. Özsu and P. Valduriez. *Principles of distributed database systems (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.

[20] C. Re, J. Brinkley, K. Hinshaw, and D. Suciu. Distributed XQuery. In *Workshop on Information Integration on the Web, VLDB*, 2004.

[21] 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.

[22] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1972.

[23] 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.

[24] Y. Zhang and P. Boncz. XRPC: interoperable and efficient distributed XQuery. In *Proc. of VLDB*, 2007.