Generating Efficient Execution Plans for Vertically Partitioned XML Databases

Patrick Kling, M. Tamer Özsu, Khuzaima Daudjee
Cheriton School of Computer Science, University of Waterloo
{pkling, tozsu, kdaudjee}@cs.uwaterloo.ca

ABSTRACT

Experience with relational systems has shown that distribution is an effective way of improving the scalability of query evaluation. In this paper, we show how distributed query evaluation can be performed in a vertically partitioned XML database system. We propose a novel technique for constructing distributed execution plans that is independent of local query evaluation strategies. We then present a number of optimizations that allow us to further improve the performance of distributed query execution. Finally, we present a response time-based cost model that allows us to pick the best execution plan for a given query and database instance. Based on an implementation of our techniques within a native XML database system, we verify that our execution plans take advantage of the parallelism in a distributed system and that our cost model is effective at identifying the most advantageous plans.

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 one 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 [19]. 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.

The complete problem of distributing an XML database is complex and comprised of many different aspects: First, a distribution model needs to be defined that is well suited to the demands of XML. Due to XML's tree structure, an XML collection can be partitioned in a largely unconstrained fashion by cutting individual document edges (e.g., [1, 9, 11]). However, the concise fragmentation schema that can be obtained by fragmenting a collection based on characteristics of data or schema is a significant asset during distributed query optimization [4, 7, 15, 16, 17]. For our work, we have, therefore, chosen a distribution model that is based on schema characteristics and query operators (similar to relational distribution models): horizontal fragmentation based on selection, vertical fragmentation based on projection, and hybrid fragmentation based on a concatenation of both operators.

The second part of the problem is distributed query evaluation. In order to perform distributed query evaluation (rather than straightforward data shipping), the query needs to be localized, yielding local query plans that can be evaluated at the sites holding the relevant data fragments. This process involves pruning unnecessary fragments to eliminate unnecessary work [11, 15]. Finally, a *distributed execution plan* needs to be generated and optimized that assembles the results of local query plans into the overall query result.

Due to the magnitude of the problem, it is unrealistic to present a complete solution to the entire problem in a single paper. Our focus in this paper is on the particular problem of generating distributed execution plans within the context of a vertically partitioned and distributed XML database system. Execution plans for horizontal and hybrid distribution will be the subject of a future paper.

Vertical partitioning allows us to store different types of data at different sites in a distributed system. Figure 1 shows an example of a vertically partitioned XML collection containing information about authors and their publications. Ignoring the highlighted nodes labeled $P_k^{i\to j}$ and $RP_k^{i\to j}$ for now, we can see that fragment f_1 contains the `author` and `agent` nodes of each document in the collection, f_2 contains the nodes pertaining to the names of authors and agents (i.e., `name`, `first` and `last` along with their text content), f_3 contains the element types `pubs`, `book` and `article` and f_4 contains the element types `chapter` and `reference`.

Consider, for example, the following query (q):

```
/author[name/last[.='Shakespeare']]//book//reference
```

In order to evaluate query q on the partitioned collection shown in Figure 1, we perform *query localization* by decomposing the query into local query plans, each of which corresponds to a single fragment. The results of these local query plans need to be joined to form the result for q. Exactly how this is done is specified in a distributed execution plan. As is well-known, there are usually many different execution plans that all produce the correct result, but whose performance varies widely. This makes choosing a distributed execution plan that is suitable for a given query and

Proceedings of the VLDB Endowment, Vol. 4, No. 1

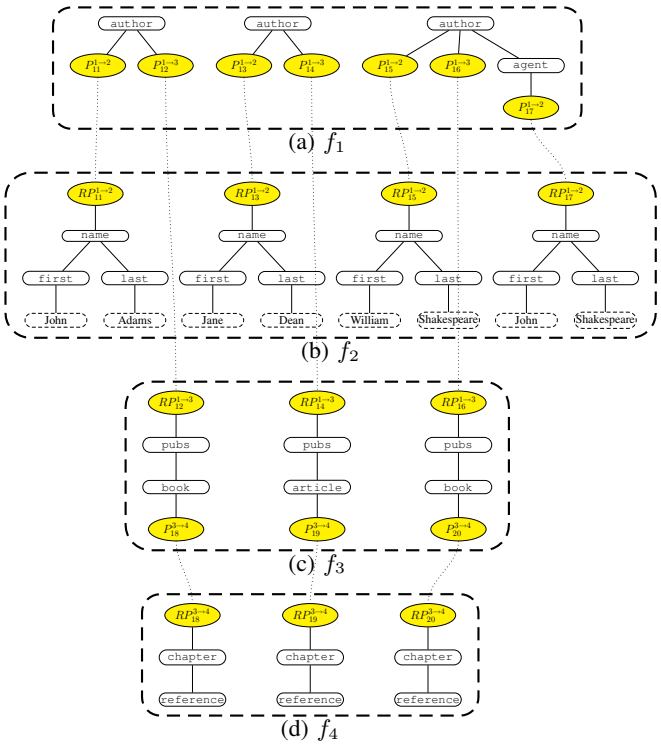

Figure 1: A vertically fragmented collection

database instance crucial for achieving good query performance. The main contribution of this paper consists of several techniques for choosing and optimizing such execution plans, allowing us to significantly improve the performance and scalability of distributed XML query evaluation.

Figure 2 shows an example of a distributed execution plan for query q, with p_i representing the local plan corresponding to fragment f_i. As can be seen, results from p_1 and p_2 are joined together first, the results of this join are then combined with results from p_3, and finally with results from p_4.

There is some existing work in the area of distributed query evaluation over XML collections. Some approaches focus primarily on minimizing the number of accesses that need to be made to each fragment of a collection [1, 11]. Another technique aims to centralize query evaluation by using a replicated index and only accessing remote fragments for certain kinds of predicates [7]. The technique presented in this paper, in contrast, is focused on finding execution plans that minimize query response time. We achieve this goal by enumerating the possible execution plans (specified within an algebraic framework) and evaluating them using a distribution-aware response time cost model.

The contributions of this paper can be summarized as follows:

- Based on our previous work on localization and local plan generation [15], we propose a technique for generating distributed execution plans within the context of a vertically partitioned and distributed XML database system.
- We analyze the benefits and drawbacks of different types of execution plans.

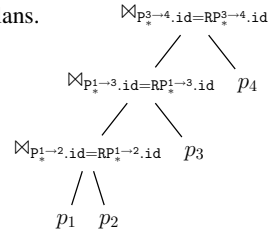

Figure 2: Distributed execution plan

- We propose a novel optimization technique that allows us to prune irrelevant subtrees in a fragment by pushing cross-fragment joins into local query plans. We ensure that this optimization does not compromise parallelism.
- We propose a cost model based on response time in a distributed system. Our focus is on composing the costs of local query plans while taking into account parallelism and the optimization techniques presented in this paper.

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 because the schema graph is never used for the validation of documents this does not pose a problem [22].

Definition 2.1 *An XML* schema graph *is defined as a 4-tuple* $\langle \Sigma, \Psi, m, \rho \rangle$ *where* Σ *is an alphabet of node types,* ρ *is the root node type,* $\Psi \subseteq \Sigma \times \Sigma$ *is a set of edges between node types 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. $m(\sigma)$ 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. Ignoring the dashed outlines for now, Figure 3 shows an example of a schema graph (corresponding to the DTD shown in Figure 7 in Appendix A).

2.2 Query Model

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 (/) 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 ". θ_s s", where s is a string constant and θ_s is either = or !=; or (iii) a numeric constraint of the form ". θ_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 [13, 18]. Therefore, solving the problem of evaluating XQ queries in a distributed fashion is an important contribution to distributed XQuery evaluation.

[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.

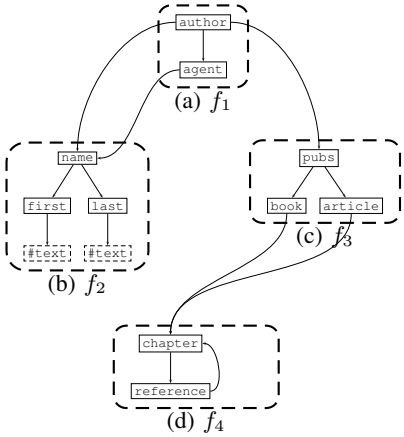

Figure 3: A vertical fragmentation schema

It is convenient to represent XQ queries as tree patterns [8, 23], which we formalize as follows:

Definition 2.2 *Let $\langle \Sigma, \Psi, m, \rho \rangle$ be the schema of the data collection. 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 $e \in E$, $\epsilon(e) \in \{\texttt{child}, \texttt{descendant}\}$ 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 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, this is beyond the scope of this paper.

The QTP depicted in Figure 4 is equivalent to query q. 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.

2.3 Vertical Fragmentation

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

Definition 2.3 *Let $\langle \Sigma, \Psi, m, \rho \rangle$ be a schema graph. A* vertical fragmentation schema *is defined by a partitioning F_Σ of the set of node types Σ such that for each fragment $f_\Sigma \in F_\Sigma$, the graph with vertices f_Σ and edges $(\Psi \cup f_\Sigma) \times f_\Sigma$ is connected.*

The dashed outlines in Figure 3 show how the node types in this schema have been fragmented into four disjoint subgraphs. Fragment f_1 consists of the node types `author` and `agent`; fragment f_2 consists of the node types `name`, `first` and `last` along with their text content; fragment f_3 consists of `pubs`, `book` and `article`; fragment f_4 includes the node types `chapter` and `reference`.

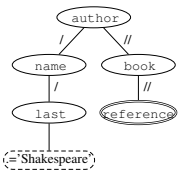

Figure 4: Query tree pattern (QTP) representation of query q

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 to another fragment f_j, we refer to f_j as a *child fragment* of f_i and to f_i as a *parent fragment* of f_j. There is exactly one fragment $f_\rho \in F_\Sigma$ that contains the root node type ρ. We refer to f_ρ 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 document 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 to fragment f_j by inserting a pair of artificial nodes $P_k^{i \to j}$ and $RP_k^{i \to j}$ into fragments f_i and f_j, respectively. $P_k^{i \to j}$ denotes a *proxy node* in fragment f_i (the originating fragment) with ID k, whereas $RP_k^{i \to j}$ denotes a *root proxy node* in fragment f_j (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 in Figure 1 has been fragmented according to the fragmentation schema in Figure 3. The proxy pair consisting of $P_{11}^{1 \to 2}$ in fragment f_1 and $RP_{11}^{1 \to 2}$ in f_2, for example, represents an edge from an `author` node in f_1 to a `name` node in f_2.

Vertical fragments generally consist of multiple unconnected pieces of XML data, which we refer to as *document subtrees*. In Figure 1, for example, fragment f_1 contains three subtrees, each of which consists of the `author` and `agent` nodes of one of the documents in the collection.

2.4 Distributed Query Evaluation

In this section we describe how queries can be evaluated on a vertically fragmented collection. We begin with a centralized QTP that represents the fragmentation-unaware query. The first step of distributed query evaluation is localization, in which the query dispatcher decomposes the centralized QTP into multiple local QTPs. Each of the resulting local QTPs references node types that correspond to a single fragment.

We then ship the local QTPs to the sites storing their corresponding fragments, where they are transformed to local algebraic query plans. The sites are free to use whatever local query evaluation strategy they choose, including navigational approaches, structural joins, and others.

After local query plans have been generated at each site, cost estimates for each local plan are reported back to the dispatcher. The dispatcher then uses these cost estimates to construct a distributed execution plan that determines how the local results are combined to produce the overall query result. The construction of such execution plans is the main contribution of this paper and will be described in Section 3. For completeness, we briefly summarize the other steps in Appendix B; they are discussed in more detail elsewhere [15].

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 3.1 *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* $G_{P'_a} \bowtie_{P_*^{i\to j}.id=RP_*^{i\to j}.id} G_{P'_b} = G_{P'}$.

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.

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 $M_{G'_P} = M_{G_{P'_a}} \cup M_{G_{P'_b}} \setminus \{P_*^{i\to j}, RP_*^{i\to j}\}$.

Figure 10 in Appendix C shows some distributed execution plans that combine the results of the local plans shown in Figure 9. In order to answer query q, we could choose any one of them, although their costs may vary. Figures 10(a) and (b) show left-deep execution plans, whereas Figure 10(c) shows a bushy execution plan. As we will discuss in the next section, some optimizations can only be performed on left-deep execution plans. Due to the parent-child relationship between local plans, it is always possible to define a left-deep execution plan.

4. OPTIMIZING DISTRIBUTED PLANS

When optimizing distributed execution plans, one area of interest is the order in which joins are performed. This problem has been studied extensively within the context of relational databases [19] and is not the focus of this paper. Instead, we focus on optimization opportunities that are unique to XML.

As discussed previously, our execution model is based on query shipping, i.e., the local plan corresponding to a fragment is evaluated at the site where that fragment is stored. In all of the distributed execution plans considered in the previous section, local plans occur only as leaves. Therefore, the evaluation of local plans is unaffected by how their results are combined by the distributed execution plan. This can lead to a scenario where a large number of local results are computed and subsequently discarded by a cross-fragment join.

Consider, for example, the execution plan shown in Figure 2. Evaluating p_3 yields results for two of the document subtrees in f_3 but the subsequent join with the local results of p_1 and p_2 discards the result derived from the subtree that does not belong to the author named Shakespeare. Similarly, p_4 yields results for all subtrees in f_4, all but one of which are subsequently discarded.

In this section, we propose two distributed optimization techniques that avoid computing unnecessary local results. As our experiments show, these techniques can greatly reduce the cost of evaluating local query plans as well as reducing the input sizes of cross-fragment joins.

4.1 Pushing Cross-Fragment Joins

The first optimization technique is based on the idea of pruning the document subtrees in a fragment by pushing cross-fragment joins from the distributed execution plan into the corresponding local query plan. We assume that each local plan (except for the root local plan, i.e., the plan corresponding to the root fragment) contains a scan of root proxy nodes in its corresponding fragment. This is a realistic assumption because the local QTPs corresponding to these plans contain a node matching these root proxy nodes as an extraction point (which means that structural-join based approaches require this scan). The document subtrees in the fragment are rooted at root proxy nodes (which means that a navigational approach visiting all subtrees also has to rely on a scan of the root proxy nodes).

By pushing a cross-fragment join to the root proxy scan, we can filter out all but those root proxy nodes that match a proxy node returned as part of the result of the parent plan. This allows us to restrict local query evaluation to the document subtrees that contain these root proxy nodes.

Since each non-root local plan p_j contains a root proxy scan as one of its leaves, we can define the *remainder plan* p'_j of p_j by removing the root proxy scan from p_j. p'_j then consumes the output of the root proxy scan: $p_j = p'_j(\text{scan}(RP_*^{i\to j}))$

We further define the *modified remainder plan* p_j^M by adding the set of attributes M to each projection in p'_j. That is, if there is a projection Π_{M_j} in p'_j, then p_j^M will contain the projection $\Pi_{M_j \cup M}$ in its place.

Based on the modified remainder plan, we can define the following rewrite, which pushes a cross-fragment join into the local plan on its right-hand side and modifies the local plan such that attributes produced by the left-hand side are preserved (a graphical representation of this rewrite is shown in Figure 11 in the Appendix):

$$G_P \bowtie_{P_*^{i\to j}.id=RP_*^{i\to j}.id} p_j \equiv p_j^{M_{G_P}} \left(G_P \bowtie_{P_*^{i\to j}.id=RP_*^{i\to j}.id} \text{scan}\left(RP_*^{i\to j}\right)\right)$$

Note that this rewrite relies on the fact that the right-hand side of the join is a single local plan. This means that, while we can rewrite all the joins in a left-deep plan, for other plan shapes, there will be some joins that cannot be rewritten.

Figure 5 shows the result of applying the rewrite to all the joins in Figure 2. Only those document subtrees in f_2 are accessed whose root proxies match a proxy in the result of p_1. Similarly, the combined results of p_1 and p_2 are used to determine the subtrees that are accessed in f_3, and the combined results of p_1, p_2 and p_3 are used to determine which subtrees are accessed in f_4. In addition to reducing the number of local results, this can also significantly improve the response times of local plan evaluation.

For performance and autonomy reasons, we assume that the sites holding individual fragments are completely independent in how they generate local plans. Therefore, we do not generally have access to local plans when optimizing a distributed execution plan.

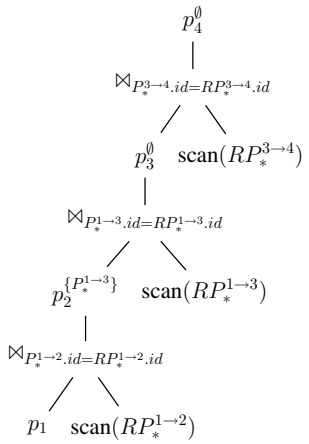

Figure 5: Plan after pushing cross-fragment joins

In order to be able to push cross-fragment joins in this scenario, we require that the sites holding individual fragments implement an API that allows

- specification of whether to generate a local plan with regular (p_i) or with modified semantics (p_i^M), and
- if modified semantics are chosen, specification of
  - the set of attributes M that are to be passed through p_i^M,
  - the cross-fragment join that is to be inserted between the root proxy scan and p_i^M, and
  - the fragment from which the left-hand side input of the cross-fragment join will be pipelined.

Using this API, which can easily be implemented at each site, it is possible to perform cross-fragment join pushing as described in this section without having access to the individual local plans.

Pushing cross-fragment joins reduces the size of intermediate results that have to be shipped and combined with results from other sites. In this respect, the effect of this technique is similar to that of using a semi-join, as is frequently done in distributed relational systems [19]. The way this reduction in intermediate result size is accomplished, however, is quite different. Unlike semi-join techniques, which require an additional inner join operation following the semi-join, our technique relies on a single join operator. More importantly, rather than incurring additional processing cost, our technique reduces the cost of local query evaluation by removing irrelevant parts of the fragment from consideration.

One reason why pushing cross-fragment joins works well in XML database systems are the complex (and therefore expensive) structural constraints in the XML query model. In relational systems, it is usually preferable to push selections past join operations in order to reduce the cost of the join. Interestingly, here, the opposite is true: it is beneficial to push a join past a collection of operators that evaluate part of a QTP (corresponding, semantically, to a selection).

4.1.1 Maintaining Parallelism

Pushing cross-fragment join operators into local query plans introduces dependencies between the local plan evaluation at different sites. In this section, we describe how we can maintain a high level of parallelism in the presence of these dependencies.

Whenever a local query plan contains a pushed cross-fragment join, execution of this query plan has to be delayed until results from query plans on the left-hand side of the pushed join have arrived. Only then can the cross-fragment join be performed and the local query plan be executed on relevant document subtrees. Waiting for the entire result of the left-hand side of the join, however, would effectively serialize the evaluation of local plans and eliminate parallelism in distributed query execution.

We address this problem using pipelined execution, such that we have to wait only for the first result tuple from the left-hand side of the join before we can start identifying the first relevant subtree. This greatly reduces the delay introduced by pushing cross-fragment joins.

To enable pipelined execution, we have to use a physical join operator that does not materialize the result of its left-hand side input. If we assume that local query results are returned ordered by their proxy IDs, we can use a simple join operator with full pipelining on both inputs. If this is not the case, we can use a hash join, which builds a hash table on its right-hand side input (i.e., the root proxy nodes) and then probes this table for each tuple from the left-hand side input. Using a hash join operator is not detrimental to pipelining because the hash table on a fragment's root proxy nodes is not query-dependent, and can easily be built ahead of time.

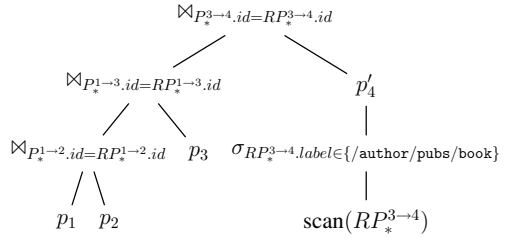

Figure 6: Plan with label path filtering

4.2 Label Path Filtering

While pushing cross-fragment joins can lead to significantly improved performance, it can only be fully applied to left-deep plans. Furthermore, waiting for the first input tuple for each cross-fragment join might result in a non-trivial reduction of parallelism in certain cases[2]. In such cases, it may be better not to push certain cross-fragment joins in the distributed execution plan. In this section, we present a technique that potentially allows us to gain at least part of the advantage of pushing cross-fragment joins with no constraints on plan shapes and no impact on parallelism.

The idea behind this technique is based on the concept of label paths. The label path of a node is defined as the sequence of node types encountered on a path from the root of a document to that node. For this optimization, we assume that each proxy pair in the collection is annotated with a label path from the root of the document it belongs to (omitting proxy/root proxy nodes on that path). $P_{18}^{3\to4}$ and $RP_{18}^{3\to4}$ in the collection shown in Figure 1, for example, would be assigned the label path `/author /pubs/book`, whereas $P_{19}^{3\to4}$ and $RP_{19}^{3\to4}$ would be assigned the label path `/author/pubs/article`. Label paths are useful not only for optimizing distributed plans but also for certain optimizations during query localization [15].

By unrolling descendant steps in the global QTP into child steps and inserting path alternatives as necessary (using a procedure described in [15]), we can obtain, for each fragment f_j, the set of label paths L_j from the root of a document to the root of a relevant document subtree in f_j. Based on this information, we can rewrite a local query plan p_j by inserting a selection between the root proxy scan and the remainder plan (see Figure 12 in Appendix D for a graphical representation):

$$p_j \equiv p_j' \left(\sigma_{RP_*^{i\to j}.label\in L_j}\left(\text{scan}(RP_*^{i\to j})\right)\right)$$

Applying this rewrite ensures that only those document subtrees are considered during local plan evaluation whose label paths are compatible with the query. Figure 6 shows the result of applying this rewrite to p_4 in the execution plan from Figure 2. In this example, the set of label paths for f_4 that are compatible with the query consists only of the single label path `/author/pubs/book`. The root proxy node $RP_{19}^{3\to4}$, on the other hand, has the label path `/author /pubs/article`. Therefore, the document subtree rooted at $RP_{19}^{3\to4}$ is not accessed during evaluation of p_4'.

5. COST ESTIMATION

In the previous sections, we described a number of alternatives for constructing distributed execution plans. To choose the best plan for a given query and collection, we need to be able to determine the cost of each candidate plan. In this section, we propose a cost model that allows us to estimate the response time of a distributed execution plan (i.e., the time it takes to completely evaluate the plan). We rely on the existence of cost models for centralized XML query evaluation and present a method for composing a cost estimate for a distributed execution plan from cost estimates for lo-

[2]Consider, for example, a case where the parent plan produces only a single tuple. Before we can execute the child plan, we have to wait for this tuple, which effectively serializes query execution.

cal plans. To determine the plan with the lowest estimated cost, we need to enumerate the plan alternatives and estimate the cost of each candidate plan. It is of course possible to reduce the size of the search space using heuristics or pruning, but that is outside the scope of this paper.

In a centralized environment, the costs of individual parts of a plan may be composed by addition. The parallelism in a distributed system makes this composition significantly more complicated. The overall response time cost of two operators that are working independently of each other in parallel, for example, is not determined by the sum of their costs but by the maximum of their costs. If there are dependencies between these operators, determining the overall cost has to take into account the time that one operator spends waiting for the other.

5.1 Local Costs

We define the following cost metrics for each local plan p_j:

- $\text{cost}(p_j)$, the response time of evaluating p_j on its corresponding fragment f_j,
- $\text{card}(p_j)$, the number of tuples returned by p_j when evaluated on the fragment f_j,
- $\text{snip}(p_j)$, the number of document subtrees accessed by p_j.

In general, we do not have access to exact values for $\text{cost}(p_j)$ and $\text{card}(p_j)$. We therefore rely on estimates of these values, which can be obtained using various cost estimation techniques that have been developed for the centralized processing of XML queries. For convenience, our notation does not distinguish between estimated cost metrics and their precise counterparts.

To determine $\text{snip}(p_j)$, we need to distinguish between two cases: If p_j is evaluated on fragment f_j, and f_j is the root fragment, then $\text{snip}(p_j) = \text{snip}(f_j)$, the number of document subtrees in fragment f_j. If f_j is not the root fragment, $\text{snip}(p_j)$ only includes the subtrees in f_j that correspond to an edge from the fragment of p_j's parent plan. Therefore, if p_j is rooted at a pattern node labeled $RP_*^{i\to j}$, $\text{snip}(p_j)$ is the number of root proxy nodes representing an edge from fragment f_i to fragment f_j ($RP_k^{i\to j}$ for all k) in f_j.

5.2 Distributed Execution Plans

Since distributed execution plans are binary trees, we need to distinguish between two cases:

1. If the distributed execution plan G_p consists of a single local plan p_i, then the cost of G_p is simply the cost of that local plan, and the cardinality of G_P is the cardinality of the local plan: $\text{cost}(G_P) = \text{cost}(p_i)$, $\text{card}(G_P) = \text{card}(p_i)$.
2. If $G_P = G_{P_a} \bowtie_{P_*^{i\to j}.id = RP_*^{i\to j}.id} G_{P_b}$, then $\text{cost}(G_P) = \max\{\text{cost}(G_{P_a}), \text{cost}(G_{P_b})\}$. This is because G_{P_a} and G_{P_b} can be evaluated in parallel and there are no dependencies between them (recall that we are optimizing response time).

   To estimate the cardinality of G_p, we define $\text{uniq}(G_{P_a}, f_j)$ to be the number of root proxies in f_j for which there is a matching proxy node in the result of G_{P_a}. We observe that $\frac{\text{uniq}(G_{P_a}, f_j)}{\text{snip}(p_j)}$ represents the fraction of root proxy nodes $RP_k^{i\to j}$ in f_j that contribute to the cross-fragment join on j. Therefore, there are $\frac{\text{uniq}(G_{P_a}, f_j)}{\text{snip}(p_j)}\,\text{card}(G_{P_b})$ results from G_{P_b} that will contribute to this join. Also, assuming independence, there are $\frac{\text{card}(G_{P_a})}{\text{uniq}(G_{P_a}, f_j)}$ tuples in the result of G_{P_a} for each root proxy node in f_j that contributes to the join.

   This allows us to estimate the cardinality of G_P as follows:

$$\begin{aligned}\text{card}(G_P) &= \frac{\text{card}(G_{P_a})}{\text{uniq}(G_{P_a}, f_j)}\,\frac{\text{uniq}(G_{P_a}, f_j)}{\text{snip}(p_j)}\,\text{card}(G_{P_b}) \\ &= \frac{\text{card}(G_{P_a})}{\text{snip}(p_j)}\,\text{card}(G_{P_b})\end{aligned}$$

   For an example, see Appendix E.1.

5.3 Pushed Cross-Fragment Joins

If we push a cross-fragment join into a local plan, we introduce a dependency that we have to account for when estimating the cost of a distributed access plan. If we let $G_P = p_j^M(G_{P_a} \bowtie_{P_*^{i\to j}.id = RP_*^{i\to j}.id} \text{scan}(RP_*^{i\to j}))$, then before we can start evaluating p_j^M, we need to wait for the first tuple produced by G_{P_a}. Assuming that G_{P_a} produces tuples at a steady rate, the first tuple is produced after a delay of $\frac{\text{cost}(G_{P_a})}{\text{card}(G_{P_a})}$. We further assume that the cost of p_j is proportional to the number of document subtrees considered. Thus, the cost of G_P can be estimated as follows:

$$\text{cost}(G_P) = \max\left\{\text{cost}(G_{P_a}), \frac{\text{cost}(G_{P_a})}{\text{card}(G_{P_a})} + \left(\text{card}(G_{P_a})\,\frac{\text{cost}(p_j)}{\text{snip}(p_j)}\right)\right\}$$

It is important to note that while pushing cross-fragment joins reduces the cardinality of a local plan, it removes only those tuples that would have been discarded during the subsequent join. The overall cardinality of the execution plan is unaffected:

$$\text{card}(G_P) = \frac{\text{card}(G_{P_a})}{\text{snip}(p_j)}\,\text{card}(p_j)$$

For an example, see Appendix E.2.

5.4 Label Path Filtering

In order to estimate the cost of a local plan with label path filtering $lp_j = p'_j\left(\sigma_{RP_*^{i\to i}.label \in L_j}\left(\text{scan}(RP_*^{i\to j})\right)\right)$, we define an auxiliary plan $l_j := \sigma_{RP_*^{i\to j}.label \in L_j}(\text{scan}(RP_*^{i\to j}))$, which simply returns all root proxy nodes whose label path is in L_j. Based on this, we can estimate the cost of lp_j by scaling according to the proportion of root proxies in f_j that is returned by l_j:

$$\text{cost}(lp_j) = \text{cost}(p_j)\,\frac{\text{card}(l_j)}{\text{snip}(p_j)}$$

Similarly, we can estimate the cardinality of lp_j:

$$\text{card}(lp_j) = \text{card}(p_j)\,\frac{\text{card}(l_j)}{\text{snip}(p_j)}$$

Since we are removing document subtrees from consideration, we also need to estimate the number of subtrees considered by lp_j:

$$\text{snip}(lp_j) = \text{card}(l_j)$$

For an example, see Appendix E.3.

6. PERFORMANCE EVALUATION

We have enhanced the native XML database system NATIX [6] with distributed capabilities and implemented our techniques within this system. This allows us to validate our approach and to perform realistic experiments. We use collections of on-line auction data generated by the XMark benchmark [21], which is a standard benchmark for evaluating XML query performance.

Our experiments are conducted on virtualized Linux machines within Amazon's Elastic Compute Cloud. We use a separate instance (providing 1.7 GB of memory, 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 Effects of Optimizations

The first experiment we perform is based on a set of queries that is designed to test a number of different cases that affect how our optimization techniques can be applied (Q1-Q5 in Table 4 in Appendix F). We evaluate these queries on an XMark collection that has been decomposed into small documents with an average size of 30 KB[3]. We scale the collection to 35 MB, 350 MB, and 3.5 GB, and fragment it according to the fragmentation schema in Figure 14 in Appendix F.

For each query and collection size, we measure the response time achieved by centralized query evaluation on a single instance holding the entire collection ("Cent"), distributed query evaluation based on a naïve, left-deep execution plan ("Dist"), distributed query evaluation with label path filtering ("Filt") and distributed query evaluation with cross-fragment join pushing ("Push"). For the experiments using optimizations we choose the plan with the lowest estimated cost. All measurements reported in this paper include the cost of result construction, shipping sub-query results between sites and shipping the overall query result to the dispatcher.

Col.	Q	Response time (s)			
		Cent	Dist	Filt	Push
35MB	Q1	0.85	0.68	N/A	0.70
	Q2	2.74	1.62	N/A	0.90
	Q3	2.71	4.76	N/A	1.00
	Q4	2.85	3.69	3.65	1.07
	Q5	0.72	2.31	1.88	0.90
350MB	Q1	6.90	6.79	N/A	4.96
	Q2	24.78	14.31	N/A	5.50
	Q3	24.73	50.16	N/A	5.63
	Q4	25.43	50.20	50.13	5.66
	Q5	5.71	54.80	42.86	5.14
3.5GB	Q1	289.47	66.93	N/A	47.21
	Q2	332.91	142.90	N/A	51.19
	Q3	331.65	522.10	N/A	51.24
	Q4	336.35	518.92	519.39	51.21
	Q5	288.56	431.42	343.99	47.55

Table 1: Performance results

The results are shown in Table 1. For queries Q1 and Q2, naïve distributed query execution (i.e., query execution without our optimizations) performs significantly better than centralized execution. This can be explained by the fact that distributed query execution can evaluate different parts of the query in parallel. For Q2, which references a larger fraction of the fragments in the system, the advantage of distributed query execution is somewhat less pronounced. This is because Q2 entails a higher overhead due to cross-fragment joins. For queries Q3, Q4, and Q5, which access the `category` fragment with its very large number of document subtrees, the overhead is so large that centralized execution performs better than naïve distributed execution. This illustrates that while distribution is useful in some cases, without further optimization, it is not a practical method for improving query performance.

With cross-fragment join pushing the picture is markedly different. For Q1 and Q2, this optimization further improves query performance by a significant margin. Even for the more adversarial queries Q3-Q5, join pushing yields a plan that outperforms centralized query execution by a factor of 6, illustrating that this is an effective technique for optimizing distributed execution plans.

Our other optimization technique, label path filtering, can only be applied to the queries Q4 and Q5 (with the other queries, there is no opportunity for filtering based on label paths). For Q4, where one fragment (shown in Figure 14(g) in Appendix F) benefits from filtering, it does not lead to a significant improvement in overall performance. For Q5, however, where label path filtering can be applied to two fragments (Figure 14(g) and (h)), we see a significant improvement when compared to naïve distributed execution, although the improvement is less pronounced than that of cross-fragment join pushing. This supports our intuition that join pushing should be preferred unless other considerations prevent us from using it (e.g., if we prefer a non-left-deep plan).

The overhead of using our cost model is small, even when exhaustively enumerating all plan alternatives. Choosing the best plan never took more than 0.01 seconds for any of the queries shown.

[3]We place each `open_auction` element into its own document along with its descendants and document subtrees referenced via ID/IDREF, yielding small documents of regular structure.

6.2 XPathMark

For the second experiment, we use a subset of the queries in the XPathMark benchmark (those that can be expressed in XQ, i.e., A1-A6 and B7, shown in Table 4 in Appendix F). Since these queries are primarily designed to evaluate the performance of evaluating XPath axes, they contain few filtering predicates and each return a large portion of the nodes in the collection as their result. While this is an important use case, we also wanted to capture the equally realistic scenario of queries that do have such filtering predicates. Therefore, we added a value predicate to each query (resulting in the selective XPathMark queries A1S-A6S and B7S). We evaluate both the original and the modified queries on a multiple-document XMark collection consisting of documents with an average size of 60 MB. We scale the collection to 120 MB, 1.2 GB, and 12 GB and fragment it into 10 vertical fragments according to the fragmentation schema shown in Figure 13 in Appendix F.

Col.	Q	Response time (s)					
		XPathMark			Selective XPathMark		
		Cent	Dist	Push	Cent	Dist	Push
120MB	A1	0.96	4.96	2.94	1.29	5.13	0.80
	A2	2.90	8.02	4.39	6.24	8.07	1.12
	A3	1.34	7.95	4.38	1.28	7.89	0.91
	A4	0.96	5.03	3.08	0.99	5.10	0.86
	A5	1.04	4.48	2.85	1.19	4.59	0.88
	A6	1.12	3.55	4.03	0.94	3.03	0.53
	B7	6.42	4.51	4.39	6.06	3.71	1.02
1.2GB	A1	8.78	46.50	26.71	12.01	45.39	6.24
	A2	28.91	77.03	40.53	62.55	73.20	8.29
	A3	12.64	76.53	39.90	11.93	72.19	6.49
	A4	8.64	46.40	27.77	8.78	44.36	6.24
	A5	9.52	42.29	25.58	10.39	39.43	6.34
	A6	9.70	31.25	36.47	8.28	27.99	3.21
	B7	63.50	40.83	40.14	59.36	31.96	8.03
12GB	A1	803.04	462.72	266.17	823.24	415.48	61.03
	A2	1021.82	763.55	402.56	1363.53	678.32	80.25
	A3	837.17	760.53	403.16	826.80	679.24	61.07
	A4	793.01	464.08	282.32	798.00	410.33	60.64
	A5	811.47	419.60	263.91	812.55	362.83	60.48
	A6	803.33	303.43	352.25	785.35	265.61	29.11
	B7	1373.06	396.50	388.80	1329.25	300.06	77.36

Table 2: Performance results

Table 2 shows the results of this experiment. We can observe a number of trends: Centralized query evaluation performs well for the smallest collection size, but its performance deteriorates rapidly as we reach the larger collection sizes.

Naïve distributed execution starts off being significantly slower than centralized execution (for the smallest collection). Once we reach the largest collection size, however, it significantly outperforms centralized execution for all queries. The margin is particularly large for the selective queries, where distributed execution benefits from smaller intermediate results and less cross-fragment join overhead.

Applying join pushing leads to a further improvement in performance and with this optimization we can achieve a speed-up factor of more than 25 compared to centralized execution. Join pushing is particularly effective for the more selective queries. This can

be explained by the fact that our cost model favours plans that exploit query selectivity on one fragment to significantly reduce the amount of data accessed in other fragments. Queries that contain highly selective predicates provide more opportunity for this and for these queries optimized distributed execution outperforms centralized execution for all collection sizes.

Together, these observations confirm the main thesis of this paper: with proper optimization, distribution is an effective technique for scaling XML database systems.

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.

Much of the existing work on distributed XML query processing assumes a distribution model without an explicit fragmentation specification [1, 2, 9, 11]. 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 [4, 7, 15, 16, 17]. 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 this work [6].

A concise graph representation of the schema of an XML collection has been used to convert XML data to relational tuples [22]. As in our work, the authors capture only the relevant aspects of the original DTD or XML Schema.

Query models similar to XQ and their connection to standard XPath and XQuery have been considered in related work [13, 18]. The representation of such queries as tree patterns is also an established technique [8, 23]. 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, but might be useful as a backend language in a distributed database system.

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 [5] and as a means to implement publish/subscribe systems [10]. Fragmentation-aware query evaluation techniques have also been used within the context of a centralized XML DBMS [14].

Much of the existing work on distributed query evaluation focuses primarily on the number of accesses that need to be made to each fragment of a collection [1, 11]. Other techniques aim to centralize query evaluation using a replicated index structure and only access remote fragments for certain kinds of predicates [7].

Techniques for pruning the set of fragments that need to be visited to answer a given query are orthogonal to the techniques presented here and can be used to further improve performance [15].

The problem of ordering and executing distributed joins is orthogonal to the problem considered here and has been studied extensively in relational systems [19].

8. CONCLUSION

In this paper, we propose a number of plan alternatives and optimizations for the distributed processing of queries in a vertically partitioned XML database system. Using a distribution-aware cost model, we identify plans that yield a significant improvement in response time, both when compared to centralized techniques and to naïve distributed approaches. Our experiments on a distributed version of a native XML database system confirm this and show that, with our optimizations, distribution is an effective technique for improving the scalability of XML query processing.

9. REFERENCES

[1] S. Abiteboul, O. Benjelloun, and T. Milo. The Active XML project: an overview. *VLDB Journal*, 17(5):1019–1040, 2008.

[2] S. Abiteboul, A. Bonifati, G. Cobéna, I. Manolescu, and T. Milo. Dynamic XML documents with distribution and replication. In *Proc. of ACM SIGMOD*, pages 527–538, 2003.

[3] 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*, pages 141–152, 2002.

[4] 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*, pages 150–163, 2006.

[5] S. Bose and L. Fegaras. XFrag: A query processing framework for fragmented XML data. In *Proc. of WebDB*, pages 97–102, 2005.

[6] M. Brantner, S. Helmer, C.-C. Kanne, and G. Moerkotte. Full-fledged algebraic XPath processing in Natix. In *Proc. of ICDE*, pages 705–716, 2005.

[7] J.-M. Bremer and M. Gertz. On distributing XML repositories. In *Proc. of WebDB*, pages 73–78, 2003.

[8] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *Proc. of ACM SIGMOD*, pages 310–321, 2002.

[9] P. Buneman, G. Cong, W. Fan, and A. Kementsietsidis. Using partial evaluation in distributed query evaluation. In *Proc. of VLDB*, pages 211–222, 2006.

[10] C.-Y. Chan and Y. Ni. Content-based dissemination of fragmented XML data. In *Proc. of ICDCS*, page 44, 2006.

[11] G. Cong, W. Fan, and A. Kementsietsidis. Distributed query evaluation with performance guarantees. In *Proc. of ACM SIGMOD*, pages 509–520, 2007.

[12] M. F. Fernàndez, T. Jim, K. Morton, N. Onose, and J. Siméon. Highly distributed XQuery with DXQ. In *Proc. of ACM SIGMOD*, pages 1159–1161, 2007.

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

[14] C.-C. Kanne, M. Brantner, and G. Moerkotte. Cost-sensitive reordering of navigational primitives. In *Proc. of ACM SIGMOD*, pages 742–753, 2005.

[15] P. Kling, M. T. Özsu, and K. Daudjee. Distributed XML query processing: Fragmentation, localization and pruning. Technical Report CS-2010-02, University of Waterloo, 2010.

[16] H. Ma and K.-D. Schewe. Fragmentation of XML documents. In *Proc. of SBBD*, pages 200–214, 2003.

[17] H. Ma and K.-D. Schewe. Heuristic horizontal XML fragmentation. In *Proc. of CAiSE*, pages 131–136, 2005.

[18] G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *J. ACM*, 51(1):2–45, 2004.

[19] M. T. Özsu and P. Valduriez. *Principles of distributed database systems (2nd ed.)*. Prentice Hall, 1999.

[20] C. Re, J. Brinkley, K. Hinshaw, and D. Suciu. Distributed XQuery. In *Workshop on Information Integration on the Web*, pages 116–121, 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*, pages 974–985, 2002.

[22] 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*, pages 302–314, 1999.

[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*, pages 54–65, 2004.

[24] Y. Zhang and P. Boncz. XRPC: interoperable and efficient distributed XQuery. In *Proc. of VLDB*, pages 99–110, 2007.

APPENDIX

A. DTD EXAMPLE

Figure 7 shows a simplified DTD corresponding to the schema graph in Figure 3.

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

Figure 7: A schema

B. LOCALIZATION AND GENERATION OF LOCAL PLANS

B.1 Localization

Localization is the process of determining which fragments are relevant for a given query and decomposing the query into sub-queries that can be evaluated on individual fragments.

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 and then transform the resulting local QTPs into algebraic query plans at the individual sites. While the details of decomposition are described elsewhere [15], we note that each local QTP only references node types from a single fragment. When wildcard nodes are encountered in the QTP, they are replaced with a special *choice node*, followed by a pattern node for each possible node type the wild-card can correspond to.

Decomposing the global QTP q from Figure 4 with respect to the fragmentation schema in Figure 3 yields the local QTPs shown in Figure 8, with q_1, q_2, q_3 and q_4 corresponding to f_1, f_2, f_3 and f_4, respectively.

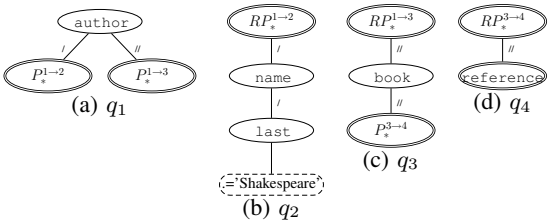

Figure 8: Local QTPs

Each cross-fragment edge in q 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.

B.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., [6, 3]). The distributed execution 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 9 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.

If q_i is a root QTP, then we call its corresponding local plan the *root local plan*. Similarly, if q_i is a child QTP of q_j, then p_i is a *child local plan* of p_j.

Each tuple returned by a local plan p_i consists of one attribute for each extraction point in q_i and one attribute for each proxy/root proxy node in q_i. We refer to the set of attributes returned by p_i as M_i. In our example, $M_1 = \{P_*^{1\to2}, P_*^{1\to3}\}$, $M_2 = \{RP_*^{1\to2}\}$, $M_3 = \{RP_*^{1\to3}, P_*^{3\to4}\}$ and $M_4 = \{RP_*^{3\to4}, \texttt{reference}\}$.

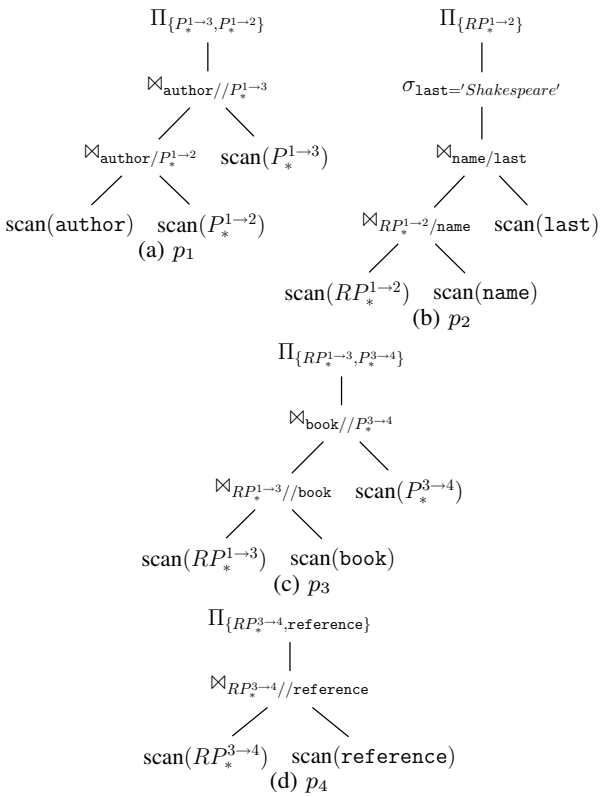

Figure 9: Local plans

C. DISTRIBUTED EXECUTION PLANS

Figure 10 shows several distributed execution plans.

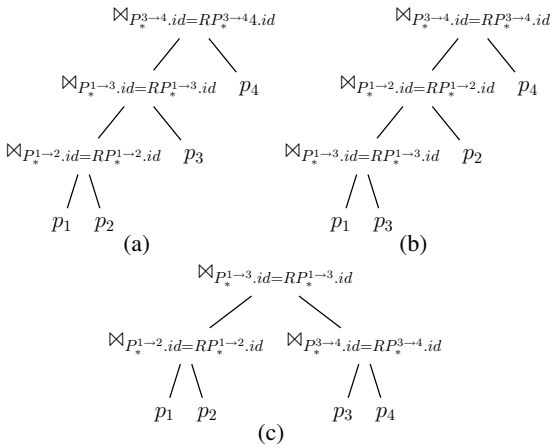

Figure 10: Distributed execution plans

D. GRAPHICAL REPRESENTATIONS OF REWRITES

Figures 11 and 12 show graphical representations of the rewrites presented in Sections 4.1 and 4.2, respectively.

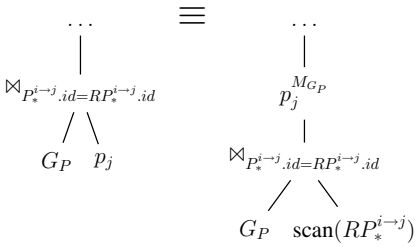

Figure 11: Cross-fragment join pushing rewrite

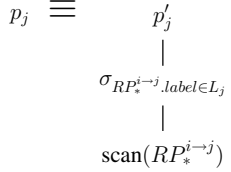

Figure 12: Label path rewrite

E. COST MODEL EXAMPLES

E.1 Distributed Execution Plans

Plan	cost	card	snip
p_1	1	3	3
p_2	2	2	4
p_3	5	2	3
p_4	10	3	3

Table 3: Local plan costs

If we assume, for example that the local plans in our example have the estimated costs, cardinalities and subtree counts shown in Table 3, then we can estimate the cost and cardinality of the execution plan G_P from Figure 10(a) as follows:

$$\begin{aligned}
\text{card}(G_{\{p_1,p_2\}}) &= \frac{\text{card}(p_1)}{\text{snip}(p_2)}\text{card}(p_2) = \frac{3}{4}2 = \frac{3}{2}\\
\text{card}(G_{\{p_1,p_2,p_3\}}) &= \frac{\text{card}(G_{\{p_1,p_2\}})}{\text{snip}(p_3)}\text{card}(p_3) = \frac{\frac{3}{2}}{3}2 = 1\\
\text{card}(G_{\{p_1,p_2,p_3,p_4\}}) &= \frac{\text{card}(G_{\{p_1,p_2,p_3\}})}{\text{snip}(p_4)}\text{card}(p_4) = \frac{1}{3}3 = 1\\
\text{cost}(G_{\{p_1,p_2\}}) &= \max\{\text{cost}(p_1), \text{cost}(p_2)\}\\
&= \max\{1,2\} = 2\\
\text{cost}(G_{\{p_1,p_2,p_3\}}) &= \max\{\text{cost}(G_{\{p_1,p_2\}}), \text{cost}(p_3)\}\\
&= \max\{2,5\} = 5\\
\text{cost}(G_P) &= \max\{\text{cost}(G_{\{p_1,p_2,p_3\}}), \text{cost}(p_4)\}\\
&= \max\{5,10\} = 10
\end{aligned}$$

E.2 Cross-Fragment Join Pushing

We can estimate the cost and cardinality of the execution plan G_P as follows:

$$\begin{aligned}
\text{cost}(G_{\{p_1,p_2\}}) &= \max\left\{\text{cost}(p_1), \frac{\text{cost}(p_1)}{\text{card}(p_1)} + \left(\text{card}(p_1)\frac{\text{cost}(p_2)}{\text{snip}(p_2)}\right)\right\}\\
&= \max\left\{1, \frac{1}{3} + \left(3\frac{2}{4}\right)\right\} = \frac{11}{6} \approx 1.83\\
\text{cost}(G_{\{p_1,p_2,p_3\}}) &= \max\Bigg\{\text{cost}(G_{\{p_1,p_2\}}), \frac{\text{cost}(G_{\{p_1,p_2\}})}{\text{card}(G_{\{p_1,p_2\}})}\\
&\quad + \left(\text{card}(G_{\{p_1,p_2\}})\frac{\text{cost}(p_3)}{\text{snip}(p_3)}\right)\Bigg\}\\
&= \max\left\{\frac{11}{6}, \frac{\frac{11}{6}}{\frac{3}{2}} + \left(\frac{3}{2}\frac{5}{3}\right)\right\} = \frac{67}{18} \approx 3.72\\
\text{cost}(G_P) &= \max\Bigg\{\text{cost}(G_{\{p_1,p_2,p_3\}}), \frac{\text{cost}(G_{\{p_1,p_2,p_3\}})}{\text{card}(G_{\{p_1,p_2,p_3\}})}\\
&\quad + \left(\text{card}(G_{\{p_1,p_2,p_3\}})\frac{\text{cost}(p_4)}{\text{snip}(p_4)}\right)\Bigg\}\\
&= \max\left\{\frac{67}{18}, \frac{\frac{67}{18}}{1} + \left(1\frac{10}{3}\right)\right\} = \frac{127}{18} \approx 7.06
\end{aligned}$$

Q1	/open_auction[initial > 200]//item//mail/from
Q2	/open_auction[initial > 200][.//author/person/name[starts-with(., 'Ry')]]//item//mail/from
Q3	/open_auction[initial > 200][.//author/person/name[starts-with(., 'Ry')]]//item//category/id
Q4	/open_auction[initial > 200][.//author/person[profile/age > 30]/name[starts-with(., 'Ry')]]//item//category/id
Q5	/open_auction[initial > 200]//author/person[starts-with(name, 'Ry')]/profile/interest/category/description
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
A1S	/site/closed_auctions/closed_auction[price > 600]/annotation/description/text/keyword
A2S	//closed_auction[price > 600]//keyword
A3S	/site/closed_auctions/closed_auction[price > 600]//keyword
A4S	/site/closed_auctions/closed_auction[price > 600][annotation/description/text/keyword]/date
A5S	/site/closed_auctions/closed_auction[price > 600][descendant::keyword]/date
A6S	/site/people/person[starts-with(name, 'Ry')][profile/gender and profile/age]/name
B7S	//person[starts-with(name, 'Ry')][profile/@income]/name

Table 4: Queries used for performance evaluation

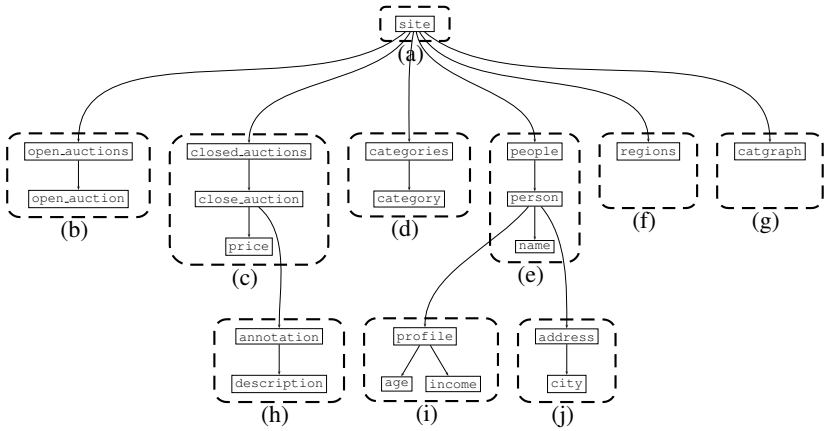

Figure 13: Fragmentation schema used in second experiment

E.3 Label Path Filtering

We can estimate the cost of $l_{p_4} = p'_4(\sigma_{RP_*^{3\to4}.label\in\{\texttt{/author/pubs/book}\}}(\text{scan}(RP_*^{3\to4})))$ as follows:

$$\begin{aligned}
\text{cost}(l_{p_4}) &= \text{cost}(p_4)\frac{\text{card}(l_4)}{\text{snip}(p_4)} = 10\frac{2}{3} = \frac{20}{3} \approx 6.67\\
\text{card}(l_{p_4}) &= \text{card}(p_4)\frac{\text{card}(l_4)}{\text{snip}(p_4)} = 3\frac{2}{3} = 2\\
\text{snip}(l_{p_4}) &= \text{card}(l_4) = 2
\end{aligned}$$

F. PERFORMANCE EVALUATION DETAILS

Table 4 shows the queries used in our experiments.

Figure 14 shows the schema used to fragment the XMark collection for our first experiment (with some node types omitted).

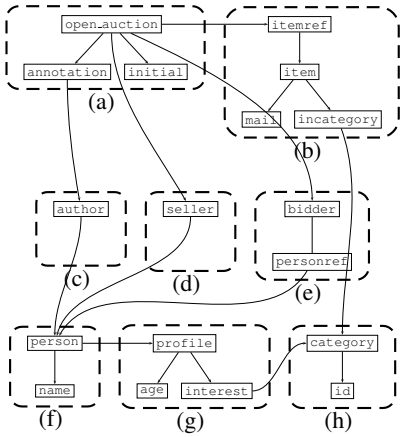

Figure 14: Fragmentation schema used in first experiment

Figure 13 shows the schema used to fragment the XMark collection for our second experiment (with some node types omitted).