# Content-based Dissemination of Fragmented XML Data

Chee-Yong Chan
Department of Computer Science
National University of Singapore
chancy@comp.nus.edu.sg

Yuan Ni
Department of Computer Science
National University of Singapore
niyuan@comp.nus.edu.sg

## Abstract

*Content-based dissemination of data using pub/sub systems is an effective means to deliver relevant data to interested data consumers. With the emergence of XML as the standard for data representation and exchange, a lot of attention has been focused on pub/sub systems for XML-based dissemination, where subscriptions are specified using more expressive XML-based languages (e.g., XPath). In this paper, we address the problem of matching XPath-based subscriptions on fragmented XML data, which is motivated by both the prevalance of resource-constrained mobile devices for accessing/monitoring data as well as by the optimization opportunities from processing data in terms of fragments. We investigate efficient strategies to schedule and optimize the evaluation of XPath-based subscriptions on XML fragments. Our experimental results not only demonstrate the effectiveness of our proposed optimizations but also reveal several interesting performance tradeoffs.*

## 1 Introduction

Content-based publish/subscribe systems provide an effective means to selectively and asynchronously disseminate information generated by *data publishers* to a large number of *data subscribers* who have preregistered their interests in specific information to some *content-based router (or message broker)* using some subscription language. Research on pub/sub systems has produced many interesting work, including efficient subscription matching algorithms (e.g., [17, 4]), subscription summarization/aggregation algorithms to reduce matching complexity and routing overhead (e.g., [6, 18, 13]), and novel pub/sub architectures to improve performance or adapt to network topological reconfigurations (e.g., [8, 14, 20]). The majority of existing pub/sub systems have typically relied on simple subscription specifications, such as keyword or "bag of words" matching, or simple comparison predicate on attribute values (e.g., Gryphon [17], Siena [5]).

The emergence of XML (eXtensible Markup Language) as a standard for information exchange on the Internet has led to an increased interest in using more expressive subscription/filtering mechanisms that exploit both the *structure* and the *content* of published XML documents. In particular, the XPath language has been adopted as a filter-specification language by a number of XML data dissemination systems [16, 9]. Due to the more expressive and complex XPath-based subscriptions, matching XML documents against such subscriptions becomes a more challenging problem, and several sophisticated algorithms have been developed to address this issue (e.g., [16, 9, 24]).

In this paper, we address the problem of matching XPath-based subscriptions on *fragmented* XML data, where the published XML data is being disseminated in terms of a collection of disjoint fragments. There are several motivations for fragmenting XML data [1, 11, 15, 12, 22]. With the popularity of employing resource-constrained mobile devices for accessing and monitoring data, there is a need for memory-efficient techniques to process queries on fragmented data. Furthermore, applications involving sensor devices typically also collect and process data in fragments. Disseminating XML data in fragments also facilitates updated data to be efficiently propagated without resending the entire document. The size of the collection of queries being matched can vary depending on the application context. A small-scale deployment can airse in specialized monitoring applications that run on mobile devices, while a large-scale scenario can arise in middleware-based applications that disseminate data to a large number of different users based on their subscriptions. While the first scenario necessarily requires the data to be fragmented for it to be processed by resource-limited devices, the second scenario can also benefit from using fragmented data as this can enable more opportunities for query optimization by exploiting the structural relationships among the fragments to minimize unnecessary and redundant processing.

While there has been some research that address general query processing issues on fragmented data [22], we are not aware of any work that examines the problem of matching boolean XPath queries on fragmented XML data. The more specialized nature of processing boolean queries on fragmented XML data opens up new opportunities for query optimization and processing. Specifically, the challenge is how to efficiently and effectively schedule and optimize the processing of the fragments so as to "short-circuit" the query evaluation as early or as much as possible by determining the evaluation result with minimal unnecessary/redundant fragment evaluations. To the best of our knowledge, our work represents the first comprehensive approach to schedule and optimize the evaluation of boolean XPath queries on fragmented XML data. Our experimental results (using both synthetic and real-life datasets) not only show that our fragmented approaches significantly outperformed the traditional non-fragmented approach, but it also reveals interesting performance tradeoffs of our proposed techniques.

## 2 Preliminaries

We focus on a commonly used subclass of XPath queries called *tree pattern (or twig) queries* that essentially supports only XPath's / and // location steps with AND-predicates. A tree pattern query is represented by an unordered rooted tree, where each node is labeled with an element name or a wildcard that is prefixed by either "/" (for a child-step) or "//" (for a descendant-step).

Given a query $q$ and an XML document $d$, a *matching of $q$ in $d$* is identified by a mapping from the nodes in $q$ to the nodes in $d$ such that both the following conditions are satisfied: (1) each mapped data node $d_i$ matches its corresponding query node $q_i$ (i.e., either $d_i$ and $q_i$ have the same element tag or $q_i$ is "*"); and (2) the structural relationships between query nodes are satisfied by their corresponding mapped data nodes. Thus, we say that $q$ *matches* $d$ if there exists at least one matching of $q$ in $d$; otherwise, $q$ does not match $d$.

Consider a node $t_i$ in a (query or data) tree $T$. We define the *prefix of* $t_i$, denoted by $prefix(t_i)$, to be the path of nodes from the root node of $T$ to $t_i$ (inclusive). We define the *minimum (maximum) height of* $t_i$, denoted by $minHt(t_i)$ ($maxHt(t_i)$), to be the length of the shortest (longest) path from $t_i$ to one of its descendant leaf nodes in $T$. Given a query node $q_j$ and a data node $d_i$, we can view $prefix(q_j)$ and $prefix(d_i)$ as a query tree and a data tree, respectively, and define the matching of $prefix(q_j)$ in $prefix(d_i)$ similarly.

When the data nodes in an XML document $d$ are partitioned into fragments, finding a matching of a query $q$ becomes more complex and requires seeking matchings of different *subqueries* of $q$ among the fragments. Given a query node $q_i$ in $q$, we define the *subquery rooted at* $q_i$, denoted by subquery($q_i$), to be the query subtree rooted at $q_i$.

**Example 2.1** Consider the XML document $d$ and query $q$ in Figs. 1(a) and (b), respectively; where $d$ is partitioned into 7 fragments indicated by the dashed regions of nodes ($f_1$ to $f_7$). It can be easily verified that the document $d$ matches the query $q$. In $f_4$, $prefix(c) = /a/m/c$, $minHt(c) = 2$, and $maxHt(c) = 2$. In $q$, $prefix(i) = /a/b/f//i$, $minHt(a) = 3$, and $maxHt(a) = 4$. Note that $subquery(/f)$ matches $f_1$, $subquery(/k)$ matches $f_3$, and $subquery(/s)$ matches $f_4$. Together with the matchings of query nodes $/b$ in $f_2$, $/c$ in $f_4$, and $/a$ and $/m$ in $f_7$, we have a matching of $q$ in $d$. ◇

## 3 Our Approach

In this section, we present our approach of processing boolean XPath queries on fragmented XML data.

### 3.1 XML Fragmentation Model

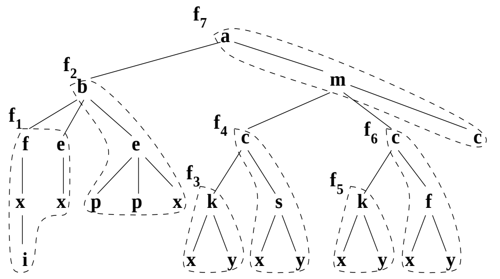
Our work assumes a very general data fragmentation model, where an XML document is partitioned into a collection of fragments that satisfy the following three properties:

P1. The fragments are *disjoint*; i.e., each document node belongs to exactly one fragment.

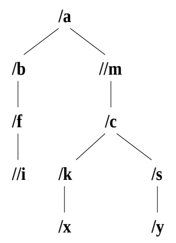
P2. The fragments are *acyclic* in the sense that whenever a fragment $f_i$ contains some data node that is an ancestor of some data node in another fragment $f_j$, then $f_j$ can not also contain a data node that is an ancestor of some data node in $f_i$.

P3. The fragments are *complete*; i.e., the original non-fragmented document can be reconstructed from the collection of fragments.

Property P1 is motivated by space-efficiency to avoid node duplication. Property P2 specifies a desirable property to ensure that document nodes are contiguous in the sense that if node $x$ is an ancestor of node $y$ and they both are stored in the same fragment, then all the nodes along the path from $x$ to $y$



(a) Fragmented XML Document



(b) Example Twig Query

| **Fragment** ID | `Prefix` | Prefix +Level |
|---|---|---|
| 1 | /f, //m | /f |
| 2 | /b, //m | /b |
| 3 | /k, //m | /k |
| 4 | /c, //m | /c |
| 5 | /k, //m | /k |
| 6 | /c, //m | /c |
| 7 | /a | – |

(c) Relevant Fragment-Query Node Matchings

**Figure 1. Fragmentation, query models, and relevant node information**

should also belong to that fragment. Property P3 is a necessary condition for correctness. These three properties are rather simple and reasonable requirements in our fragmentation model; and they are indeed satisfied by various strategies that have been proposed for fragmenting XML data [23, 19]. In general, a fragment can consist of a forest of subtrees of nodes (e.g., $f_1$ in Fig. 1(a) contains two subtrees rooted at nodes $f$ and $e$). Furthermore, a subtree in a fragment does not necessarily correspond to a complete subtree in the XML document (e.g., the subtree rooted at node $b$ in $f_2$ in Fig. 1(a) is partitioned between fragments $f_1$ and $f_2$).

In order to guarantee Property P3 (i.e., fragment completeness), it is necessary to maintain some additional header information for each fragment to enable the fragments to be "stitched" together to reconstruct the original XML document. In addition to ensuring completeness, note that the header information associated with the fragments can also be exploited for query processing as it actually provides some partial structural information about the fragments and their relationships.

## 3.2 Overview of Processing XML Fragments

Our approach of processing boolean XPath queries on fragmented XML data consists of three main steps.

1. **Identify relevant fragments**. The first step is to make use of the collection of fragment header information to identify a set of "relevant" matchings to determine for each fragment. The goal is to minimize both the number of relevant matchings as well as the number of fragments to be evaluated.
2. **Schedule fragment evaluations**. The second step is to determine an order in which to process the fragments. The goal is to "short-circuit" the query evaluation as early or as much as possible.
3. **Evaluate subqueries on fragments**. This step deals with how to efficiently optimize and process the set of relevant subqueries associated with each fragment.

## 3.3 Fragment Header Information

In this section, we discuss two annotation schemes for representing fragment header information, namely, `Prefix` and `Prefix+Level`. These schemes have different space-performance tradeoffs.

**`Prefix` Annotation.** The first method stores information about the path leading to the root node of each subtree in a fragment. Specifically, each fragment is associated with the following header information: (1) a unique identifier for the fragment; and (2) for each subtree (rooted at a data node $d_i$) in the fragment, its prefix given by $prefix(d_i)$. For example, the header information for $f_3$ in Fig. 1(a) is the tuple $(3, /a[1]/m[2]/c[1]/k)$. Note that for convenience, we have used positional predicates in $prefix(d_j)$ to distinguish among distinct data paths that share the same sequence of element tag names; other means of achieving this purpose (e.g., assigning each node with a unique nodeID attribute value) can be used as well.

**`Prefix+Level` Annotation.** The second method is a simple extension of `Prefix`, that additionally records $maxHt(r)$ for each subtree rooted at node $r$ in a fragment. For example, the header information for $f_3$ in Fig. 1(a) is $(3, /a[1]/m[2]/c[1]/k,1)$. As we shall explain in Section 3.4, the additional precomputed information turns out to be very effective in improving query evaluation as it can avoid unnecessary computations.

## 3.4 Identifying Relevant Fragments

To improve matching efficiency, our goal is to determine for each query node $q_j$, the set of "relevant" fragments that can potentially contain matchings of $q_j$. Informally, a fragment $f_i$ is said to be *relevant* for a query node $q_j$ (or equivalently, $f_i$ is a relevant fragment for $q_j$) if based on the fragment header information, $f_i$

contains some subtree that could contain a matching of $subquery(q_j)$. For notational convenience, we use $\mathcal{R}$ to denote the set of all relevant fragment-query node pairs between a given fragmented document $d$ and a twig query $q$; i.e., $(f_i, q_j) \in \mathcal{R}$ iff fragment $f_i$ is relevant for query node $q_j$. In the following, we elaborate on how relevant fragment-query node pairs are identified for the two types of fragment header information.

`Prefix` **Annotation.** With the `Prefix` header annotation scheme, $(f_i, q_j) \in \mathcal{R}$ if there exists a subtree rooted at $r$ in $f_i$ such that $prefix(q_j)$ matches $prefix(r)$. For example, in Fig. 1, since $prefix(/b)$ matches $prefix(b)$ in $f_2$, we have $(f_2, /b) \in \mathcal{R}$. However, when $q_j$ is a descendant-step, the relevance checking needs to be more elaborate. For example, although $prefix(//m)$ does not match $prefix(b)$ in $f_2$ (in Fig. 1), it is incorrect to conclude that there can not be a matching of subquery(//m) in $f_2$. Indeed, $prefix(//m)$, which is given by $/a//m$, is equivalent to (/a/m $\cup$ /a//*//m); and it is clear that $/a//*$ matches $prefix(b)$ in $f_2$. To correctly capture both the cases of relevance matching, we define the *extended prefix* of a query node $q_j$, denoted by $eprefix(q_j)$, as follows:

$$eprefix(q_j) = \begin{cases} prefix(q_j) & \text{if } q_j \text{ is a child-step,} \\ prefix(q_k)//* & \text{if } q_j \text{ is a descendant-step \&} \\ & q_k \text{ is the parent node of } q_j, \\ //* & \text{otherwise.} \end{cases}$$

Therefore, $(f_i, q_j) \in \mathcal{R}$ iff there exists a subtree rooted at $r$ in $f_i$ such that $prefix(q_j)$ or $eprefix(q_j)$ matches $prefix(r)$.

`Prefix+Level` **Annotation.** With the additional maximum height information, the `Prefix+Level` annotation scheme provides a more precise definition of relevance. Specifically, $(f_i, q_j) \in \mathcal{R}$ iff there exists some subtree rooted at $r$ in $f_i$ such that (1) $prefix(q_j)$ or $eprefix(q_j)$ matches $prefix(r)$ and (2) $minHt(q_j) \leq maxHt(r)$.

**Example 3.1** Consider again the fragment $f_2$ and query $q$ in Fig. 1. With the `Prefix` annotation, $f_2$ is relevant for both query nodes $/b$ and $//m$. However, with the `Prefix+Level` annotation, $f_2$ is relevant only for query node $/b$. The reason that $f_2$ is not relevant for $//m$ is because $maxHt(b) = 2$ which is less than $minHt(q_m) = 3$. Fig. 1(c) shows all the relevant query node matchings for query $q$ in Fig. 1(b) under both `Prefix` and `Prefix+Level` annotations. $\diamond$

## 3.5 Scheduling Fragment Evaluations

To optimize the processing of the fragments, it is important to schedule the fragment evaluations so as to minimize the processing of unnecessary fragments (i.e., fragments whose evaluations could be skipped without affecting the query's result). In this section, we present five policies for scheduling fragment evaluations.

**Topological Scheduling, T**. This policy evaluates a fragment $f_i$ before another fragment $f_j$ if some node in $f_i$ has an edge pointing to some node in $f_j$.

**Reverse-Topological Scheduling, R**. This is the reverse of topological scheduling, where fragment $f_i$ is evaluated before fragment $f_j$ if some node in $f_j$ has an edge pointing to some node in $f_i$.

**Most-Specific Scheduling, S**. The intuition for this policy is that a fragment $f_i$ is more likely to contain some query node matching than another fragment $f_j$ if $f_i$'s prefix is more "specific" than $f_j$'s prefix in terms of matching some query node's prefix. This is captured by the *specificity* of a fragment $f_i$, denoted by $s(f_i)$, which is given by $s(f_i) = \max_{(f_i,q_j)\in\mathcal{R}}\{|prefix(q_j)|\}$, where $|prefix(q_j)|$ denote the number of non-wildcard steps in $prefix(q_j)$. A fragment with a larger specificity value is processed earlier.

**Maximal-Matching Scheduling, M**. The intuition for this policy is that a fragment that contains more relevant subtrees has a higher chance of producing a matching. This notion is captured by the *maximal-matching metric* of a fragment $f_i$, denoted by $m(f_i)$, which is given by $m(f_i) = \sum_{(f_i,q_j)\in\mathcal{R}} |\{s_{i,k} \mid s_{i,k}$ is a subtree in $f_i, s_{i,k}$ is relevant for $q_j\}|$. Fragments are processed in non-increasing maximal-matching values.

**Most-Critical Scheduling, C**. This policy is optimized for non-matching queries by trying to process earlier "critical" query nodes that can be potentially matched only in very few fragments. Let $F(q_j)$ denote the set of fragments that can potentially contain a matching for query node $q_j$; and let $Q(f_i)$ denote the set of query nodes that can potentially be matched in fragment $f_i$. A query node $q_j$ is defined to be *critical* if $|F(q_j)| \leq |F(q_k)|$ for each query node $q_k$ in $q$. A fragment $f_i$ is defined to be critical if there exists some critical query node in $Q(f_i)$. We define the *criticality of a critical fragment* $f_i$, denoted by $c(f_i)$, as follows:

$c(f_i) = (\sum_{q_j\in Q(f_i)} |F(q_j)|)/|Q(f_i)|$. Then, critical fragments are processed before non-critical ones; and critical fragments are processed in non-descending order of their criticality values.

**Example 3.2** Consider the document $d$ and query $q$ in Fig. 1. In terms of `Prefix+Level` annotation, $\mathcal{R} = \{(f_1, /f), (f_2, /b), (f_3, /k), (f_4, /c), (f_5, /k), (f_6, /c)\}$. For scheduling $S$, $f_2$, which has the smallest *specificity*, is skipped since a matching is found after processing the other fragments. For scheduling $M$, the order of fragment evaluation is arbitrary due to the equal *maximal-matching* value. To illustrate the policy $C$, we replace the query node $//i$ with $//j$ to make $q$ becomes a non-matching query. The order of the *criticality* values of the 6 relevant fragments is $f_1 = f_2 < f_5 = f_6 < f_3 = f_4$. After processing $f_1$ and $f_2$, $//j$ has not been matched and none of the remaining fragments are relevant with $//j$; thus the query will not be matched. $\diamond$

### 3.6 Evaluating Subqueries on Fragments

The processing of a data fragment entails the simultaneous matching of the set of subqueries relevant for that fragment. This requires the detection and maintenance of various matching data nodes as the data nodes in a fragment are parsed and processed. The matching of subqueries in fragments involves two challenges. Firstly, since the data subtrees in a fragment are not necessarily complete as different parts of a subtree might be distributed over several fragments, the matching algorithm for fragments needs to be generalized to handle partial matchings of subqueries. Secondly, since the fragments are not necessarily evaluated in a "contiguous" manner, the presence of partial matchings in various fragments need to be maintained to enable the partial matchings to be "combined" to detect complete matchings.

We have extended an existing approach for matching multiple queries, XTrie [9], to handle both these requirements. The modified algorithm is able to process multiple subqueries on fragments, and it returns both the matched subqueries as well as the matched portions of partially matched subqueries. To enable partial matchings to be combined to form complete matchings, a subquery matching at query node $q_i$ is propagated to its closest branching ancestor node, say $q_j$, to facilitate the detection of a subquery matching at $q_j$ when more subquery matchings are detected for the various child subtrees of $q_j$. The details of these extensions can be found in the full version of the paper [10].

**Example 3.3** Consider the document $d$ and query $q$ in Fig. 1 and assume that `Prefix+Level` annotation is used. Suppose $f_4$ is first processed with the relevant subquery $subquery(/c)$. Since there is no complete matching of $subquery(/c)$ in $f_4$, the algorithm returns the partial matching $/c/s/y$, which means that $subquery(/s)$ is matched. Now, suppose the next fragment to be processed is $f_3$, which will result in $subquery(/k)$ being matched. From these two subquery matchings with different fragments, the algorithm detects a complete matching of $subquery(/c)$. $\diamond$

### 3.7 Processing Multiple Queries

In this section, we briefly discuss the additional extensions required to process multiple queries simultaneously on fragmented XML data. There are two main differences from processing a single query. Firstly, the identification of relevant matchings for the various subqueries of different queries can be processed efficiently by exploiting any common prefixes among the subqueries to avoid unnnecessary processing overhead. Secondly, for the fragment scheduling schemes, the scheduling metric value for each query-dependent scheduling policy needs to be generalized to consider the subqueries from all queries. As an example, the *specificity* value of each fragment should be the sum of the *specificity* values for all relevant queries.

## 4 Dynamic Optimizations

In this section, we present two novel optimizations to further speed up the evaluation of twig queries on fragmented XML data by eliminating certain relevant evaluations. Our new optimizations utilize dynamic information about the processed fragments to eliminate certain yet-to-be-processed relevant evaluations without affecting correctness.

**Eliminating Redundant Evaluations, $+$.** This optimization is based on using the existence of some matching in a processed fragment to eliminate certain relevant evaluations in yet-to-be-processed fragments. Given subquery($q_i$) and $q_j$ is the nearest ancestor branch node of $q_i$, suppose subquery($q_i$) is matched and the data node that matches $q_j$ is $n_j$, then for all subquery($q_k$) in which $q_k$ is some descendant node of $q_i$(including $q_i$), if $n_j$ matches some prefix of $q_k$, subquery($q_k$) is *redundant*, since the matching of subquery($q_k$) will finally form a redundant matching of subquery($q_i$) under the same data node $n_j$.

**Eliminating Unnecessary Evaluations, $-$.** This optimization is based on using the absence of some matching in a processed fragment to eliminate certain relevant evaluations in yet-to-be-processed fragments. Suppose $q_b$ is a branch node in a tree pattern query $q$ and $n_b$ is the data node in the document that matches $q_b$. If there exists some branch of subquery($q_b$) that is not matched in the subtree of $n_b$, then for all subquery($q_d$), in which $q_d$ is some descendant node of $q_b$ and subquery($q_d$) will be evaluated at the subtree rooted at $n_b$, it is considered as *unnecessary*, since it has been guaranteed that no matching of subquery($q_b$) at $n_b$ does not exist.

**Example 4.1** Consider the fragmented document $d$

and query $q$ in Fig. 1. After processing $f_1$, the evaluation of query node $/f$ at $f_2$ becomes redundant since there is already a complete matching of $subquery(/f)$ in $f_1$ under the same data node $b$ in $f_2$. After processing $f_6$, the evaluation of query node $/k$ with $f_5$ becomes unnecessary since (1) there is no matching of $subquery(/s)$ in $f_6$ and (2) there are no other descendant fragments of $f_6$ (besides $f_5$) that could potentially provide a matching of $subquery(/s)$. Thus, even if there is a matching of $subquery(/k)$ in $f_5$, this will not yield a complete matching of $subquery(/c)$. $\diamond$

## 5 Related Work

The most related work to ours is the paper by Bose and Fegaras on XFrag [22] which examines processing XQuery queries on fragmented XML data based on the *hole-filler* model. In their work, the processing of an operator can be suspended while waiting for a missing fragment to arrive. In Active XML [21], XML documents are embedded with service calls to generate additional data during query processing, which can be considered as an alternative fragmentation model that is suitable for pull-based web-service applications.

There exist a body of work that focused on XML-based pub/sub systems [16, 24, 9], which assume that data is disseminated as complete XML documents while our emphasis is on exploiting fragmented data for optimized filtering. There are also a number of papers addressing the query processing on fragmented XML data [11, 7, 3] in distributed or P2P environments. All these work are optimizing the query evaluations at different sites, which is different from our work where queries are evaluated at one place.

Another relevant line of work examines how to fragment XML documents [15, 19, 12, 1]. These papers are complementary to our work which focuses on processing queries on fragmented data.

## 6 Performance Study

We have conducted a comprehensive performance study using both synthetic and real-life datasets. Our results demonstrate the efficiency of our query processing approach on fragmented data and the effectiveness of our scheduling strategies and optimizations. In our experiments, we observed that the total size of header information is no more than 1% of the none-fragmented document.

### 6.1 Experimental Testbed and Methodology

**Data Sets:** The synthetic data XMark [2] ($D_{XMark}$) and the real-life data from DBLP ($D_{DBLP}$) are used. The size of the document for $D_{XMark}$ is 11.5MB by setting the scale factor of XMark as 0.1. Documents are fragmented using Natix's algorithm [23] which controls the number of data nodes per fragment using a threshold $t$. By setting $t$ to 5000, we obtained 34 fragments for $D_{XMark}$. We have constructed eight XPath queries for $D_{XMark}(Q_1 - Q_4$ are matching queries, and $Q_5^- - Q_8^-$ are non-matching queries). The queries represent various properties of twig queries, including "//" location step, single path queries and tree pattern queries. For the specific queries, please refer to the full version paper [10]. A set of random queries were generated using YFilter's XPath generator [24].

| **Algorithm** | Values |
|---|---|
| Scheduling | `R`, `T`, `S`, `M`, `C` |
| Header annotation | `P(Prefix)`, `PL(Prefix+Level)` |
| optimizations | none, $+$, $-$, $\{+, -\}$ |

**Table 1. Different Algorithms Options**

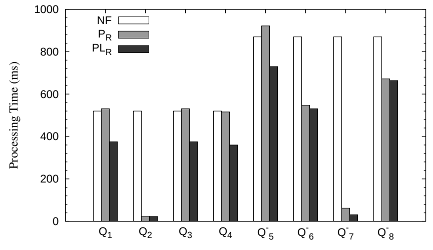
**Algorithms:** Table 1) lists the variations for different fragmented approaches. $A_y^x$ is used to denote a fragmented approach, where $A \in \{\mathtt{P}, \mathtt{PL}\}$ represents the fragment header annotation scheme used; $y \in \{\mathtt{R}, \mathtt{T}, \mathtt{S}, \mathtt{M}, \mathtt{C}\}$ (as denoted in Section 3.5) represents the scheduling policy used; and $x$ represents the set of dynamic optimizations used. For example, $PL_S^{+,-}$ denote the fragmented approach using the `Prefix+Level` annotation scheme, most-specific scheduling policy, and both dynamic optimizations. $NF$ is used to represent the non-fragmentation approach.

For the scheduling policies, not all fragments may be available at the moment of processing. An alternative mechanism is that we only wait for the arriving of $k$ fragments to perform the scheduling. Here, $k$ is called *scheduling window size*.

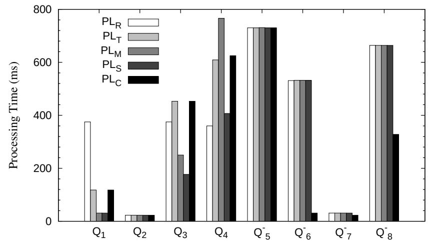
Our performance metric is the query processing time (in ms) including both the time to identify relevant fragments and the time to schedule and process the fragments. Our experiments were conducted on a 3 GHz Intel Pentium IV machine with 1 GB of main memory running Windows XP; and all algorithms were implemented using C++.

### 6.2 Experimental Results

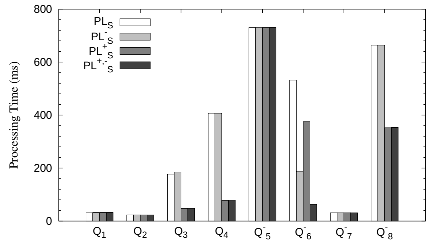
Due to the space limitation, we omit the results on $D_{DBLP}$ which showed similar trends as $D_{XMark}$.



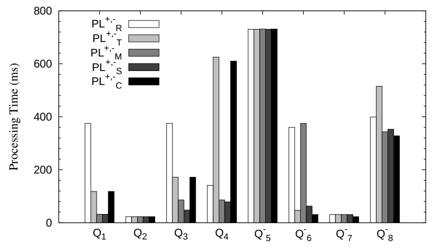
(a) NF vs. $\mathrm{P}_R$ vs. $\mathrm{PL}_R$



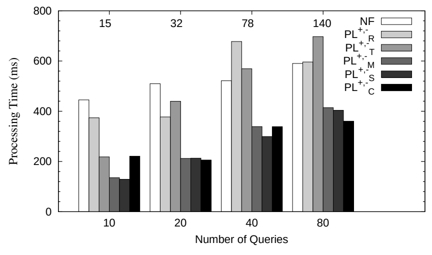
(b) Comparison of schedulings



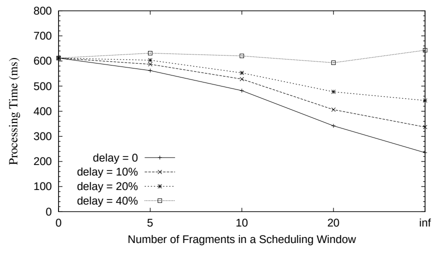
(c) Effect of optimizations on $S$



(d) Optimizations on different schedulings



(e) Effect of number of queries



(f) Effect of transmission delay, $PL_S^{+,-}$, $Q_{40}$

**Figure 2. Experimental Results, $D_{XMark}$**

**Fragmented vs. Non-fragmented Approaches.** Fig. 2(a) compares the `NF` against $P_R$ and $PL_R$ for various queries on $D_{XMark}$. The results show that $P_R$ and $PL_R$ generally outperformed `NF`; in particular, for $Q_2$, $PL_R$ reduces the processing time of `NF` by 95%. The performance improvement is due to the fact that fragmented approaches are able to process the fragments selectively based on relevant information. Even for the case that each fragment is relevant, the performance of $P_R$ is still competitive with `NF`, which means that the time to process header information is trivial. We also observe that $PL_R$ is consistently more efficient than $P_R$ since `Prefix+Level` annotation is able to exploit the additional $maxHt()$ information to prune off more non-relevant fragments. We observe that the improvement of $PL_R$ over $P_R$ is more significant for $D_{DBLP}$ than $D_{XMark}$ since $D_{DBLP}$ is shallower than the $D_{XMark}$ such that $maxHt()$-based pruning has more opportunities to prune. We will not include `Prefix`-based methods in subsequent experimental graphs.

**Comparison of Scheduling Policies.** Fig. 2(b) compares five different scheduling policies (i.e. `R`, `T`, `M`, `S`, `C`) for various queries on $D_{XMark}$. The results show that `S` is generally the most competitive for matching queries (except for $Q_4$); while `C` is generally the best policy for non-matching queries. The reason for the relatively weaker performance of the `S` for $Q_4$ is that many redundant evaluations with high specificities for the two most specific branches in $Q_4$ delay the matching of the remaining branch. However, as shown later, when the redundant elimination optimization is also applied, $PL^+_S$ outperforms the other policies for $Q_4$.

**Effect of Dynamic Optimizations.** Figs. 2(c) consider the impact of the optimizations on most specific scheduling $S$. We observe that for tree pattern queries (e.g., $Q_3$, $Q_4$, $Q^-_6$, and $Q^-_8$), dynamic optimizations is effective to eliminate redundant/unnecessary evaluations. Our results also reveal that the unnecessary evaluation optimization is less significant than the redundant evaluation optimization due to the fact that the $D_{XMark}$ data provides more opportunities for eliminating redundant evaluations. We have applied the optimizations on other schedulings, which showed the similar effectiveness. However, eliminating redundant evaluations achieves best improvement for scheduling $S$, since $S$ is likely to find matchings early such that $+$ has more chances to eliminate redundant evaluations. We also observe that combining both optimizations achieves the best performance. Fig. 2(d) compares the effect of the combined optimizations with various scheduling policies on $D_{XMark}$. The results show that for matching queries, $PL_S^{+,-}$ offers the best performance, while for non-matching queries, $PL_C^{+,-}$ gives the best performance.

**Effect of Number of Queries** Fig. 2(e) demonstrates the performance for multiple queries by varying the number of queries from 10 to 80. The y-axis is the average processing time for all queries in the set. The numbers above the bars are the time to generate header annotations. As the number of queries increases, the time to find relevant queries increases. However, as aforementioned the time to determine the relevant queries can be further improved by sharing the processing of common prefix in the queries. We also observe that the improvement becomes smaller as the number of queries increases, since it is likely that more fragments are relevant. However, it shows that even for $Q_{80}$, $PL_S^{+,-}$ and $PL_C^{+,-}$ can still outperform `NF`, since they help to find matching or non-matching queries earlier such that more redundant/unnecessary queries can be eliminated, which can further help to skip the processing of more fragments.

**Effect of Scheduling Window Size and Transmission Delay** In this section, we vary the scheduling window size $k$ to illustrate the effect, where $k = inf$ is the case that performs scheduling on all fragments. The bottom line in Fig. 2(f) shows the effect of *scheduling window size* $k$ on scheduling $S$ without considering any delay. The similar trends are observed for other scheduling. With the increasing of $k$, a better scheduling strategy is obtained such that the query processing time is reduced. Then we increase the transmission delay in the query processing. We control delay for each fragment as a percentage of the time to parse the fragment, which is varied from 0, 10%, 20%, to 40%. From Fig. 2(f), we observe that as the transmission delay increases from 0 to 20%, the improvement of larger $k$ becomes smaller, since more time is spent to wait for the fragments. And for the percentage of 40%, if $k$ is further increased to $inf$, the processing time is worse than the case $k = 0$, since the gain by scheduling on larger set of fragments is compensated by the time to wait for the available of all these fragments.

## 7 Conclusions

In this paper, we have provided the first comprehensive study on processing XPath boolean queries *directly* on fragmented XML documents without reconstructing the original documents. Our experimental results based on both synthetic and real-life datasets demonstrate the effectiveness of our processing and optimization strategies with a performance improvement of up to a factor of 20 over the conventional approach of processing non-fragmented documents. Among the various fragment header annotation schemes, scheduling policies, and evaluation optimizations that we considered, the $PL_S^{+,-}$ combination turns out to be the best approach for evaluating matching queries, while the $PL_C^{+,-}$ combination turns out to be the best approach for evaluating non-matching queries.

## References

[1] W3C (2001) XML Fragment Interchange. http://www.w3.org/TR/xml-fragment/.

[2] XMark. http://monetdb.cwi.nl/xml/index.html.

[3] A. Bonifati et al. XPath lookup-queries in P2P networks. In *WIDM*, 2004.

[4] A. Carzaniga, A.L. Wolf. Forwarding in a content-based network. In *SIGCOMM*, 2003.

[5] A. Carzaniga, D.S. Rosenblum, A.L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3), 2001.

[6] A. Carzaniga, M. Rutherford, A.L. Wolf. A routing scheme for content-based networking. In *INFOCOM*, 2004.

[7] A. Deshpande et al. Cache-and-query for wide area sensor databases. In *SIGMOD*, 2003.

[8] A. Riabov et al. New algorithms for content-based publication-subscription systems. In *ICDCS*, 2003.

[9] C.-Y. Chan et al. Efficient filtering of XML documents with XPath expressions. *VLDB Journal*, 11(4), 2002.

[10] C.-Y. Chan, Y. Ni. *Content-based dissemination of fragmented XML data.* http://www.comp.nus.edu.sg/ niyuan/fullversion.pdf.

[11] D. Suciu. Distributed query evaluation on semistructured data. *ACM TODS*, 27(1), 2002.

[12] Eugene Y. C. Wong, Alvin T. S. Chan, Hong-Va Leong. Efficient management of XML contents over wireless environment by XStream. In *SAC*, 2004.

[13] G. Li, S. Hou, H-A. Jacobsen. A unified approach to routing, covering and merging in publish/subscribe systems based on modified binary decision diagrams. In *ICDCS*, 2005.

[14] G. Picco, G. Cugola, Amy L. Murphy. Efficient content-based event dispatching in the presence of topological reconfiguration. In *ICDCS*, 2003.

[15] J-M. Bremer, M. Gertz. On distributing XML repositories. In *WebDB*, 2003.

[16] M. Altinel, M.J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *VLDB*, 2000.

[17] M. K. Aguilera et al. Matching events in a content-based subscription system. In *PODC*, 1999.

[18] P. Triantafillou, Andreas A. Economides. Subscription summarization: a new paradigm for efficient publish/subscribe systems. In *ICDCS*, 2004.

[19] R. Bordawekar, O. Shmueli. Flexible workload-aware clustering of XML documents. In *XSym*, 2004.

[20] R. Zhang, Y. Hu. HYPER: a hybrid approach to efficient content-based publish/subscribe. In *ICDCS*, 2005.

[21] S. Abiteboul et al. Dynamic XML documents with distribution and replication. In *SIGMOD*, 2003.

[22] S. Bose, L. Fegaras. XFrag: a query processing framework for fragmented XML data. In *WebDB*, 2005.

[23] T. Fiebig et al. Anatomy of a native XML base management system. *VLDB Journal*, 11(4), 2002.

[24] Yanlei Diao et al. Path sharing and predicate evaluation for high-performance XML filtering. *ACM TODS*, 28(4), 2003.