THE UNIVERSITY *of* EDINBURGH

Edinburgh Research Explorer

Partial Evaluation for Distributed XPath Query Processing and Beyond

OPEN ACCESS

Partial Evaluation for Distributed XPath Query Processing and Beyond

GAO CONG, Nanyang Technological University
WENFEI FAN, University of Edinburgh and Harbin Institute of Technology
ANASTASIOS KEMENTSIETSIDIS, IBM T.J. Watson Research Center
JIANZHONG LI and XIANMIN LIU, Harbin Institute of Technology

This article proposes algorithms for evaluating XPath queries over an XML tree that is partitioned horizontally and vertically, and is distributed across a number of sites. The key idea is based on partial evaluation: it is to send the whole query to each site that partially evaluates the query, in parallel, and sends the results as compact (Boolean) functions to a coordinator that combines these to obtain the result. This approach possesses the following performance guarantees. First, each site is visited at most twice for data-selecting XPath queries, and only once for Boolean XPath queries. Second, the network traffic is determined by the answer to the query, rather than the size of the tree. Third, the total computation is comparable to that of centralized algorithms on the tree stored in a single site, regardless of how the tree is fragmented and distributed. We also present a MapReduce algorithm for evaluating Boolean XPath queries, based on partial evaluation. In addition, we provide algorithms to evaluate XPath queries on very large XML trees, in a centralized setting. We show both analytically and empirically that our techniques are scalable with large trees and complex XPath queries. These results, we believe, illustrate the usefulness and potential of partial evaluation in distributed systems as well as centralized XML stores for evaluating XPath queries and beyond.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Query Processing*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Distributed XML documents, XPath queries, parallel query processing

1. INTRODUCTION

Partial evaluation (aka program specialization Jones [1996]) has been studied in the context of programming languages as a general optimization technique. Intuitively, given a function $f(s, d)$ and part of its input s, partial evaluation is to specialize $f(s, d)$ with respect to the known input s. That is, it performs the part of f's computation that

Fig. 1. Investment company clientele.

Fig. 2. Tree partition and fragment tree.

depends only on *s*, and generates a partial answer, that is, a (residual) function f' that depends on the as yet unavailable input *d*.

Partial evaluation has been proved useful in a variety of areas including compiler generation, code optimization and dataflow evaluation (see [Jones 1996] for a survey). The last of these bears sufficient connections with distributed query evaluation. This suggests that it is worth investigating its use in parallel query processing. We focus here on the evaluation of XPath queries.

Consider the XML tree T shown in Figure 1, which represents the *clientele* of an investment company. For each *client* the company stores her *name* and the *country* where she resides. Furthermore, it stores the *broker(s)* whom the *client* is using and the *market(s)* in which the *client* trades, through the *broker(s)*. For each such *market*, the company stores the *code*, buying price *buy*, and quantity *qt* of *stock(s)* that the *client* owns. In practice such trees are often partitioned into a number of subtrees, or *fragments*, and are distributed over the Internet for geographical or administrative reasons [Bremer and Gertz 2003], a setting commonly found in e-commerce, Web services, P2P systems [Bonifati and Cuzzocrea 2006], XML data integration [Abiteboul et al. 2002], or while managing large-scale network directories [Jagadish et al. 1999]. In Figure 1, we use dashed lines to show one possible fragmentation. The fragment marked as F_0, which includes the root of the tree, might be stored in the investment company's local US server (site S_0). However, for tax reasons, trade data for Canadian customers might have to be stored in a Canada-based server. Therefore, the fragment of the tree marked as F_3 is stored in a remote Canadian site S_3. Similarly, the NASDAQ market might require that all its own trade data are only remotely accessed and only through recognized brokers for security concerns. Therefore fragments F_2 and F_4 again need to be stored in site S_2 outside our investment company. The fragmentation and distribution of the tree T are depicted in Figure 2. In spite of the reasons that lead to fragmentation, conceptually this is still a single XML tree over which we would like to pose queries.

Now consider $Q = [//stock/code/text() = "GOOG"]$, a Boolean XPath query posed at site S_0. When evaluated at the root of the tree T, the query returns a single truth value,

true if and only if there is a client trading GOOG stock. A naive way to evaluate the query is by first shipping fragments F_1, F_2, F_3, F_4 to site S_0, assembling them with F_0 into a tree, and then evaluating Q on the tree. This incurs excessive overhead of data shipping, and worse still, may not be doable since data at some sites is not allowed to be shipped to S_0 for privacy or security reasons. Another approach is to employ a sophisticated algorithm for evaluating XPath queries in a centralized system [Koch 2003]. This would require a single depth-first traversal of T, visiting each node once, something we cannot expect to improve upon. However, this seemingly "optimal" approach would visit sites $S_0, S_1, S_2, S_1, S_0, S_2, S_0, S_3, S_0$ in that order, visiting S_0 four times and S_1 twice.

We approach this problem by introducing an algorithm, referred to as ParBoX, based on partial evaluation. The algorithm partially evaluates the whole query Q, in parallel, on each fragment of the tree. Since a fragment contains only part of the tree, this partial evaluation of Q on each fragment results in a partial answer to the query, which is a Boolean expression with variables. The partial answers are all collected to a coordinator site and are composed, yielding the final answer to Q.

The ParBoX algorithm has several desirable properties. (a) Each site is visited only once, irrespectively of the number of fragments stored there. (b) The communication cost is bounded by the size of the query and the number of fragments, and is independent of the size of the XML document. (c) The total amount of computation performed at all sites holding a fragment is comparable to the computation of the optimal centralized algorithm over the whole tree. (d) The algorithm does not impose any condition on how the XML documents are partitioned, what the sizes of these fragments are, or how they are assigned to sites.

Next consider a data-selecting XPath query $Q' = //broker[//stock/code/text() = "GOOG"]/name$. Along the same lines as before, one wants to send Q' to every site and partially evaluate Q'. However, as opposed to Boolean queries, the final answer to Q' is a set of elements rather than a single Boolean value. It is challenging to identify precisely what elements are in the final answer at each site before they are shipped to the coordinator. Furthermore, for data-selecting XPath queries even centralized evaluation algorithms require two traversals of the tree [Koch 2003], instead of a single pass.

We develop algorithms and optimization methods for evaluating generic data-selecting XPath queries. The proposed techniques guarantee that each site is visited at most twice, irrespectively of the number of fragments stored there (recall that sophisticated centralized algorithms would require two passes of XML trees to evaluate data-selecting queries). Moreover, the algorithm has the same worst-case computational complexity as ParBoX and optimal communication cost, in spite of its support of more complex data-selecting queries. In particular, each site ships to the coordinator only elements that are certainly in the answer of a query, such that the union of these partial answers is the answer of the query.

As remarked earlier, the principles of partial evaluation have been proved useful in a variety of areas. It is therefore not a coincidence that the algorithms we propose here are both flexible and generic, and applicable to a variety of contexts. We illustrate some of this flexibility by adapting our distributed algorithms to work in a centralized environment with a single processing-thread. Lacking parallelism in the centralized setting, our main focus shifts from evaluation times and the benefits of our techniques manifest themselves in the form of both reduced I/O and small memory requirements (since fragments are only accessed a fixed number of times and they can be as small as the amount of memory allocated to us). In that sense, our algorithms address some of the main limitations of existing centralized XML query engines like Saxon, which suffer from the requirement that the processed XML document must fit in memory. Of course, other popular and efficient centralized XML query processing systems, like

MonetDB, do not suffer from such memory limitations. However, this efficiency comes at a cost. For a system like MonetDB to process a query over an XML document, this document must be loaded into MonetDB, a process that is often several orders of magnitude more expensive that the cost of querying the document. Our query evaluation techniques do not impose such requirement since we can read and evaluate a query over a (new) XML document directly from the file system without any preprocessing. As such, our techniques are particularly useful in domains like life sciences (e.g., biology), astronomy, and even for the management of typical XML documents corresponding to Microsoft Office files (since PowerPoint presentations, Word files, and Excel spreadsheets are all currently stored as XML). In all these domains, the management of XML documents is file-system centric and no *traditional* XML data management systems is yet in place (since nonexpert users often find these latter systems to be hard to use and maintain).

To illustrate the generality of our algorithms, we also go beyond the confines of our own implementation, and show how easily our algorithms can be adapted to work into the widely popular Map-Reduce framework [Dean and Ghemawat 2004]. Indeed, as a proof of concept, we offer a Map-Reduce algorithm for evaluating Boolean XPath queries, based on partial evaluation.

To verify the effectiveness of the proposed partial-evaluation techniques in all these contexts and for all the variations, we conduct an extensive experimental study. The experimental results demonstrate that the proposed algorithms scale well with large XML data and with complex XPath queries, in both the distributed setting and the centralized context.

Taken together, the main contributions of this article are as follows. Based on partial evaluation, we propose

(1) an algorithm for evaluating Boolean XPath queries,
(2) algorithms for evaluating data-selecting queries on XML data that is partitioned and distributed, with provable performance guarantees on the number of visits, data shipment and total computation. In addition, we provide
(3) a MapReduce algorithm for evaluating Boolean XPath queries, based on partial evaluation,
(4) algorithms for processing XPath queries on very large data that cannot be loaded to memory, and
(5) an experimental study, in both distributed and centralized settings, that verifies the effectiveness and efficiency of the partial evaluation techniques.

Organization. Section 2 presents XML tree partition and the class of XPath queries considered in this article. Sections 3 and 4 introduce algorithms for evaluating Boolean XPath queries and data-selecting queries in distributed settings, respectively, and develop optimization techniques. The MapReduce algorithms is presented in Section 5. Sections 6 provides algorithms for XPath evaluation on large data in the centralized setting. An experimental study is presented in Section 7, followed by related work in Section 8 and topics for future work in Section 9.

2. PRELIMINARIES

We next discuss partitioning of XML documents and present the class of XPath queries studied here.

2.1. XML Tree Partitioning

We consider settings in which an XML tree T is partitioned into a set $\mathcal{F}_T$ of disjoint *subtrees* of T, or *fragments*. Each fragment $F_i \in \mathcal{F}_T$ can be stored in a different site.
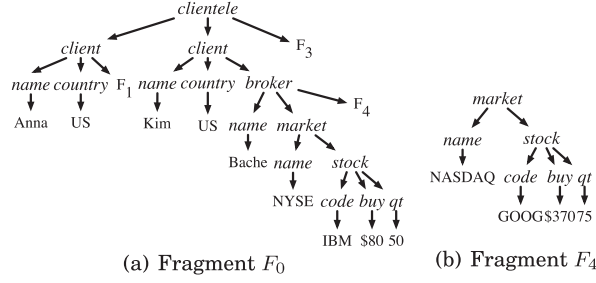
(a) Fragment F_0 (b) Fragment F_4

Fig. 3. Example fragments.

We do not impose any constraints on the partitioning: we allow arbitrary "nesting" of fragments. Fragments can appear at any level of the tree, and different fragments may have different sizes (in terms of number of nodes). Furthermore, we do not impose any constraints on how the fragments are distributed.

As an example, the tree in Figure 1 consists of five fragments, $\mathcal{F}_T = \{F_0, F_1, F_2, F_3, F_4\}$, each fragment represented by a dotted polygon.

Observe that a partitioning of a tree T induces another tree, called a *fragment tree*, which depicts the relationship between the different fragments of T. We use $\mathcal{F}_T$ to denote both the set of fragments of a tree T and the induced fragment tree. It will be clear from the context which of the two notions we refer to. The fragment tree $\mathcal{F}_T$ for the tree T of Figure 1 is shown to the right of Figure 2. We call *root* fragment, the fragment at the root of the fragment tree that also contains the root of tree T. In Figure 2, the root fragment is fragment F_0.

Given two fragments F_j and F_k, we say that F_k is a *subfragment* of F_j if F_k is a child of F_j in the fragment tree. If F_k is a subfragment of F_j then there exists a node $v \in F_j$ such that the root node w of F_k is a child of v in the original tree T. For example, in Figure 1, fragment F_2 is a subfragment of F_1, which, in turn, is a subfragment of fragment F_0. In the original tree T, node *broker* of fragment F_1 is the parent of root node *market* of fragment F_2.

Each fragment is possibly stored in a different site, as shown to the right of Figure 2. For example, fragment F_3 is stored in site S_3 while both fragments F_2 and F_4 are in site S_2. Given the distribution of fragments, we need to maintain the relationship between a fragment and its subfragments so as to preserve the structure of the original tree T. To this end, given a fragment F_j and its subfragment F_k, we add a virtual node in F_j, which we label as F_k, in place of the missing tree fragment F_k. Under normal circumstances, while traversing fragment F_j, we know that if we reach the virtual node F_k, we need to *"pass control"* to the site holding fragment F_k in order to continue the traversal of the tree. For example, in Figure 3(a) fragment F_0 has virtual nodes representing fragments F_1, F_3, and F_4. While traversing fragment F_0 in site S_0, when we reach virtual node F_4, we know that the control for the traversal of the tree must pass to site S_2 holding fragment F_4, shown in Figure 3(b).

We refer to a fragment that has no subfragments as a *leaf* fragment. In Figure 3(b), fragment F_4 is a leaf fragment and therefore it has no virtual nodes.

2.2. XPath

We consider a class of XPath queries, denoted by $\mathcal{X}$, that is defined as follows:

$$Q := \epsilon \mid A \mid * \mid Q//Q \mid Q/Q \mid Q[q],$$
$$q := Q \mid q/\textit{text}() = \textit{str} \mid q/\textit{val}() \; \textit{op num} \mid \neg q \mid q \wedge q \mid q \vee q,$$

where Q is a *path expression* defined in terms of the empty path ϵ (*self*), label A (tag), wildcard $*$, child '/', the *descendant-or-self-axis* '//', and *qualifier* $[q]$. In the qualifier q, *str* is a string constant, *op* stands for one of the arithmetic comparison operators $=, \neq, <, \leq, >, \geq$, *num* is a number, and $\neg, \wedge, \vee$ are the Boolean negation, conjunction and disjunction operators, respectively.

For example, to get the *names* of *brokers* through which GOOG stocks are purchased, but no YHOO stocks, we issue query

Q_1: *//broker[//stock/code/text() =*"GOOG" $\wedge \neg$(*//stock/code/text() = *"YHOO")]/*name*.

At a *context node* v in an XML tree T, the evaluation of a query Q at v, denoted by $\mathsf{val}(Q, v)$, yields the set of nodes of T reachable via Q from v. On a *centralized* XML tree T, that is, when T is not decomposed and distributed, $\mathsf{val}(Q, r)$ can be computed in $O(|T|\,|Q|)$ time [Gottlob et al. 2002], where r is the root of T.

The class $\mathcal{X}$ of queries subsumes twig queries [Bruno et al. 2002; Ramanan 2002]. Although $\mathcal{X}$ supports only the self, child and descendant XPath axes, this usually suffices since the majority of XPath queries use the downward axes [Ives et al. 2002].

We also consider the class of Boolean XPath queries in $\mathcal{X}$, denoted by $\mathcal{X}_{\mathrm{BL}}$. A query in $\mathcal{X}_{\mathrm{BL}}$ is of the form $\epsilon[q]$ or simply $[q]$, where $[q]$ is defined as before. For example, for the query Q_1 given earlier, $[Q_1]$ is a query in $\mathcal{X}_{\mathrm{BL}}$.

At a context node v in an XML tree T, a query $[q]$ evaluates to a truth value, denoted by $\mathsf{val}(q, v)$, indicating whether q is satisfied at v, that is, whether $\epsilon[q]$ is empty. Boolean XPath queries are commonly used, for example, in publish-subscribe systems [Altinel and Franklin 2000] and LDAP directories [Smith and Howes 1997].

We convert each query Q in $\mathcal{X}$ to a normal form $\beta_1/\ldots/\beta_n$, where β_i is one of A, $*$, // or $\epsilon[q]$. Function $\mathsf{normalize}(Q)$ inductively normalizes, in linear-time, a query Q as follows:

$\mathsf{normalize}(\epsilon)\qquad\quad = \epsilon$; similarly for '$*$', '//' and A;
$\mathsf{normalize}(Q_1/Q_2) = \mathsf{normalize}(Q_1)/\mathsf{normalize}(Q_2)$;
$\mathsf{normalize}(Q[q]) = \mathsf{normalize}(Q)/\epsilon[\mathsf{normalize}(q)]$;
$\mathsf{normalize}(Q/\text{text}() = \text{'}str\text{'}) = \mathsf{normalize}(Q)/\epsilon[\text{text}() = \text{'}str\text{'}]$;
$\mathsf{normalize}(Q/\text{val}() = \text{'}num\text{'}) = \mathsf{normalize}(Q)/\epsilon[\text{val}() = \text{'}num\text{'}]$;
$\mathsf{normalize}(q_1 \wedge q_2) = \mathsf{normalize}(q_1) \wedge \mathsf{normalize}(q_2)$; similarly for $q_1 \vee q_2$ and $\neg q_1$;
$\mathsf{normalize}(\epsilon[q_1]/\ldots/\epsilon[q_n]) = \epsilon[\mathsf{normalize}(q_1) \wedge \ldots \wedge \mathsf{normalize}(q_n)]$;

where the last rule is to combine a sequence of ϵ's into one. Hereafter, we only consider normalized $\mathcal{X}$ queries.

By striking out all the qualifiers from a normalized query $Q = \beta_1/\ldots/\beta_n$, we get what we refer to as the *selection path* of Q, the form $\eta_1/\ldots/\eta_n$, where η_i is A, ϵ, $*$, or //. For example, the selection path of query Q_1 is *//broker/name*.

For reasons that will become clear in the next section, we decouple the evaluation of qualifiers for each query Q, from the evaluation of its selection path. We use a vector-based representation of our queries. More specifically, we use a vector $\mathsf{SVect}(Q)$ to store the prefixes of the selection path $\eta_1/\ldots/\eta_n$, such that $\mathsf{SVect}(Q)[i]$ indicates the query $\eta_1/\ldots/\eta_i$. Obviously, vector $\mathsf{SVect}(Q)$ is linear in the size of Q. We use another Boolean vector $\mathsf{QVect}(Q)$ to store the list of all subqueries of the qualifiers of Q. We sort $\mathsf{QVect}(Q)$ in a topological order such that for any subqueries q_1, q_2, if q_1 is a subquery of q_2 then q_1 precedes q_2 in $\mathsf{QVect}(Q)$. Again, vector $\mathsf{QVect}(Q)$ is linear in the size of Q.

Example 2.1. Consider query

$Q = $ *client[country/text() = *"US"]/*broker[market/name/text() = *"NASDAQ"]/*name*,

which returns the *name* of brokers of US clients that trade in the NASDAQ market. Then normalize(Q) is

client/ε [country/ε [text()= "US"]]/broker/ ε [market/name/ε [text() = "NASDAQ"]]/name

We decouple the selection path */client/broker/name* of the query from the qualifiers [∗/ε [country/ε [text() ="US"]] and [∗/ε [market/name/ε [text() = "NASDAQ"]]. Then, vectors SVect(Q) and QVect(Q) are as follows:

SVect(Q) = [q_1, q_2, q_3] where
$q_1 = client,$ $q_2 = q_1/broker,$ $q_3 = q_2/name$
QVect(Q) = [$q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9$], where
$q_1 = country,$ $q_2 = [text()="US"],$ $q_3 = q_1/ε[q_2],$ $q_4 = ∗/ε[q_3],$ $q_5 = name,$
$q_6 = [text()="NASDAQ"],$ $q_7 = q_5/ε[q_6],$ $q_8 = market/q_7,$ $q_9 = ∗/ε[q_8],$

where the first four entries in QVect(Q) are for the first qualifier, while the others are for the second. Note that the subqueries of a Boolean vector correspond to the elements of the Boolean vector.

3. DISTRIBUTED EVALUATION OF BOOLEAN QUERIES

We first focus on evaluating Boolean XPath queries in the distributed setting. Consider an $\mathcal{X}_{BL}$ query q submitted to a site S, referred to as the *coordinating* site. Query q is to be evaluated at the root of a partitioned and distributed XML tree T.

A naïve evaluation strategy is to first collect all the fragments of tree T identified by the fragment tree $\mathcal{S}_T$ at the coordinating site, and then use a *centralized* algorithm, for instance, the algorithm of Gottlob et al. [2002]. We refer to this approach as NaiveCentralized. This approach is efficient once the coordinating site gets all the data. However, the price is that large fragments are sent over the network, *each* time that a query is evaluated. In addition, since the coordinating site must store these fragments during the evaluation of q, the benefits gained by distributing large XML trees over a network are alleviated. Moreover, privacy and security concerns may prevent certain sites from releasing their data to another site.

A better solution, referred to as NaiveDistributed, is to customize a centralized evaluation algorithm so that it works in a distributed fashion. We know that a Boolean XPath query can be evaluated on a single site via a single traversal of the tree T. We can use the information from the fragment tree $\mathcal{S}_T$ to perform a distributed bottom-up traversal of tree T. To do this, we need to pass certain information between the sites in the fragment tree $\mathcal{S}_T$, as the distributed computation is passed forth and back from a fragment F_i in site S_j to one of its subfragments F_k in site S_l. For example, consider the fragments and fragment tree in Figure 2. As we compute the query for fragment F_0 in site S_0, we need to pass the control of computation to fragment F_1 in site S_1. At the same time, site S_0 has to wait for this computation to finish before it continues with fragment F_4 in site S_2.

While this distributed algorithm does not require any transmission of fragments, it has two shortcomings. First, for a site S_i to finish processing its fragment F_j, it has to wait for all the other sites that hold subfragments of F_j to finish. Therefore, the distributed algorithm actually follows a sequential execution and does not take advantage of parallelism. Second, a site is visited as many times as the number of fragments stored in it. In our example, site S_2 needs to be visited twice, since it holds fragments F_2 and F_4. For each of these visits, site S_2 has to exchange a number of messages,

---

Procedure ParBoX

Input: An $\mathcal{X}_{\mathrm{BL}}$ query q and a partitioned, distributed tree T
Output: The Boolean value ans of q over T

1. $q_L := \mathsf{QVect}(q)$;
2. $\mathcal{S}_T :=$ retrieve the fragment tree of tree T;
3. **for** each site S_i in the fragment tree $\mathcal{S}_T$ **do**
4. $execute(S_i, \mathsf{evalQual}, q_L)$;
5.   **for** each fragment F_j stored in S_i **do**
6.   annotate $\mathcal{S}_T$ with $(QV_{F_j}, QCV_{F_j}, QDV_{F_j})$;
7. $ans := \mathsf{evalST}(\mathcal{S}_T)$;

Fig. 4. ParBoX algorithm executed at coordinating site.

resulting in increased network traffic, and its processor has to switch context once per fragment.

To overcome these limitations, we propose next the Parallel Boolean XPath (ParBoX) evaluation algorithm, based on partial evaluation. The ParBoX Algorithm *guarantees* the following: (1) Each site is visited only *once*, irrespectively of the number of fragments stored in it. (2) Query processing is performed in parallel, on all the participating sites. (3) The total computation on all sites is comparable to what is needed by the best-known centralized algorithm. (4) The total network traffic is determined by the size of the query rather than the XML tree.

3.1. The ParBoX Algorithm

The algorithm is shown in Figure 4 and Figure 5. It is initiated at the coordinating site, which, we assume w.l.o.g. to be the site storing the root fragment of the tree T over which the $\mathcal{X}_{\mathrm{BL}}$ query q is evaluated. The algorithm consists of three stages:

Stage 1. Initially (lines 1–2 of Procedure ParBoX in Figure 4), the coordinating site uses the fragment tree $\mathcal{S}_T$ to identify which other sites hold fragments of tree T. In our example, the coordinating site S_0 uses the fragment tree in Figure 2 to identify sites S_1, S_2 and S_3.

Stage 2. The coordinating site along with all the sites identified in the first stage evaluate, in parallel, the same input query q on all their assigned fragments (Procedure evalQual, Figure 5). Since fragments are parts of the tree T, query evaluation on each fragment returns a *partial* answer to the query q.

Stage 3. Finally (lines 5–7 of Procedure ParBoX), the coordinating site collects the partial answers from all the participating sites and all the fragments; it then composes them to compute the answer to query q.

We now describe the two crucial components of the algorithm: (a) how to compute partial answers in parallel (in Stage 2), and (b) how to assemble the partial answers to obtain the answer to query q (in Stage 3).

Partial evaluation. There is a *dependency* relation between partial evaluation processes for the query q on different fragments of the XML tree T. To see this, consider an efficient evaluation of q over T via a single bottom-up traversal of T. During the traversal, at each node v we compute the values at v of all the subqueries $\mathsf{QVect}(q)$ of query q, where $\mathsf{QVect}(q)$ is described in Section 2.2. This computation requires the (already computed) values of the $\mathsf{QVect}(q)$ subqueries at the children of v. At the end of the traversal, the answer to query q is computed by using the values of the $\mathsf{QVect}(q)$

Procedure evalQual

Input: A list q_L of sorted (sub-)queries
Output: A vector $F[S_i][F_j]$ for each fragment F_j of site S_i

1. **for** each subtree F_j assigned to S_i **do**
2. $(QV_{F_j}, QCV_{F_j}, QDV_{F_j}) = \mathsf{bottomUp}(root(F_j), q_L);$
3.   send $(QV_{F_j}, QCV_{F_j}, QDV_{F_j})$ to the coordinating site;

Procedure bottomUp

Input: A node v and a list q_L of (sub-)queries
Output: Vectors QV_v, QCV_v and QDV_v of formulas for node v

1. **for** each child w of v **do**
2. $(QV_w, QCV_w, QDV_w) := \mathsf{bottomUp}(w, q_L);$
3.   **for** each query q_i in q_L **do**
4. $QCV_v(q_i) := \mathsf{compFm}(QCV_v(q_i), QV_w(q_i), OR);$
5. $QDV_v(q_i) := \mathsf{compFm}(QDV_v(q_i), QDV_w(q_i), OR);$
6. **for** each query q_i in q_L from left to right **do**
7.   **case** q_i **of**
8. $\epsilon: QV_v(q_i) := 1;$
9. $*: QV_v(q_i) := 1;$
10. $A: QV_v(q_i) := compareString(label(), A);$
11. $text() = str: QV_v(q_i) := compareString(text(), str);$
12. $q_j/\epsilon[q_k]: QV_v(q_i):=\mathsf{compFm}(QV_v(q_j), QV_v(q_k), AND);$
13. $q_j/q_k: QV_v(q_i):=\mathsf{compFm}(QV_v(q_j), QCV_v(q_k), AND);$
14. $//q_j: QV_v(q_i) := QDV_v(q_j);$
15. $q_j \vee q_k: QV_v(q_i) := \mathsf{compFm}(QV_v(q_j), QV_v(q_k), OR);$
16. $q_j \wedge q_k: QV_v(q_i):=\mathsf{compFm}(QV_v(q_j), QV_v(q_k), AND);$
17. $\neg q_j: QV_v(q_i) := \mathsf{compFm}(QV_v(q_j), NULL, NEG);$
18. $QDV_v(q_i) := \mathsf{compFm}(QV_v(q_i), QDV_v(q_i), OR);$

Procedure compFm

Input: Two formulas f_1 and f_2 and an operator op
Output: The composed formula $ans := f_1 \; op \; f_2$

1. **case** $(isFormula(f_1)), isFormula(f_2))$ **of**
2. (c0) (0,0): **if** $op = NEG$ **then** $ans := \neg f_1;$
3.   **else** $ans := f_1 \; op \; f_2;$
4. (c1) (0,1): **if** $op = AND$ **then**
5.   **if** $f_1 = true$ **then** $ans := f_2;$ **else** $ans:=false;$
6.   **elseif** $op = OR$ **then**
7.   **if** $f_1 = true$ **then** $ans := true;$ **else** $ans:=f_2;$
8. (c2) (1,0): **if** $op = NEG$ **then** $ans := \neg f_1;$
9.   **elseif** $op = AND$ **then**
10.   **if** $f_2 = true$ **then** $ans := f_1;$ **else** $ans:=false;$
11.   **elseif** $op = OR$ **then**
12.   **if** $f_2 = true$ **then** $ans := true;$ **else** $ans:=f_1;$
13. (c3) (1,1): **if** $op = NEG$ **then** $ans := \neg f_1;$
14.   **elseif** $op = AND$ **then** $ans := f_1 \wedge f_2;$
15.   **elseif** $op = OR$ **then** $ans := f_1 \vee f_2;$

Fig. 5.   ParBoX algorithm executed at participating site.

queries at the root of the tree. More specifically, the answer to q is the value of the last query in $\mathsf{QVect}(q)$.

Consider now Figure 3 which shows the fragments of the XML tree in Figure 1. These are the trees over which the sites must compute the query q. Recall that in

these fragments some of the leaves are virtual nodes, that is, they are pointers to other fragments that reside in other sites. For example, in fragment F_1 there is a virtual leaf node marked by F_2, while fragment F_0 has three virtual leaves, one for fragment F_1, one for F_3 and one for F_4. In accordance to the strategy given before, at each site S and for each fragment F, we need to perform a bottom-up evaluation of query q. But how can we compute, during the traversal, the values of the QVect(q) subqueries at the virtual nodes? The values of the QVect(q) subqueries are unknown for these nodes and, under normal circumstances, until we learn these values from another site we cannot proceed with the evaluation.

We propose a technique to decouple the dependencies between partial evaluation processes and thus avoid unnecessary waiting, by introducing Boolean variables, one for each missing value of each QVect(q) subquery at each virtual node. Using these variables, the bottom-up evaluation procedure is given in Figure 5. Procedure bottomUp considers the root of a fragment F_j and a list q_L of subqueries that is essentially the QVect(q) of the initial query q. Recursive calls of the procedure are used to perform the bottom-up traversal of the tree F_j (line 2). At each node v, the procedure computes the "values" of q_L at v and stores the results of the computation in a vector QV_v which is of the same size as list q_L. Note that these "values" are actually *Boolean formulas* with those variables introduced at the virtual nodes. The computation of the q_L values at v requires the values of q_L computed in the children and descendants of v. We save these values (lines 3–5) by maintaining only two additional vectors, namely vectors QCV_v and QDV_v, that are of the same size as vector QV_v. Intuitively, for each subquery q' in q_L, $QCV_v(q')$ is true if and only if there exists some child u of v such that $QV_u(q')$ is true, and similarly, $QDV_v(q')$ is true if and only if either $QV_v(q')$ is true or there exists some descendant w of v such that $QV_w(q')$ is true.

Given a query $q_i \in q_L$ at a node v, the computation of the value of q_i depends on the structure of q_i. We consider different cases (lines 6–17) of the structure based on the normal form given in Section 2.2. For example, if query q_i is of the form *text*() = *str* (line 10), then its value is *true* if the text content of node v is equal to the string *str*, and is *false* otherwise. More interesting is the case where q_i is of the form $*/q_j$ (line 11). Then, the value of q_i at node v is equal to the disjunction of the values of query q_j at the child nodes of v. Due to the recursive evaluation, the value of the disjunction has already been computed in $QCV_v(q_j)$. Similarly, when q_i is $//q_j$ (line 13), the value of q_i at node v is the disjunction of $QV_v(q_j)$ and $QDV_w(q_j)$'s for the children w of v, which have again been computed by the bottom-up processing order following the list q_L of subqueries. Finally, when q_i is of the form $q_j \wedge q_k$, the value of q_i is the conjunction of the values of queries q_j and q_k. If queries q_j and q_k had simple Boolean values as answers, then this computation would be trivial. However, we note that a distinguishing characteristic of the procedure is that variables are part of our evaluation. Therefore, we compose Boolean values with variables or compose Boolean variables with other Boolean variables to create more complex formulas. Procedure compFm is responsible for composing, for each query, the truth values and/or formulas necessary to compute the value of the query. Depending on the value of the operator op it computes $f_1 \ op \ f_2$, which yields either a Boolean value or a Boolean formula.

Example 3.1. Consider an $\mathcal{X}_{\mathrm{BL}}$ query $[q]$, where q is defined as //stock[code/text()="YHOO"]. By normalizing $[q]$ (see Section 2.2) we get:

QVect($[q]$) = $[q_1, q_2, q_3, q_4, q_5, q_6, q_7]$, where

q_1=code, q_2=[text()="YHOO"], q_3=$q_1/\epsilon[q_2]$, q_4=stock, q_5=$q_4/\epsilon[q_3]$, q_6=$//q_5$, q_7=$*/\epsilon[q_6]$

Evaluating the (sub-)queries in QVect($[q]$) for the F_1 fragment nodes results in the following QV_v vectors:

— $QV_{name} =< 0, 0, 0, 0, 0, 0, 0 >$
— $QV_{F_2} =< x_1, x_2, x_3 = x_1 \wedge x_2, x_4, x_5 = x_4 \wedge x_3, x_6 = dx_5, x_7 = x_6 >$
— $QV_{broker} =< 0, 0, 0, 0, 0, dx_5, dx_5 >$

We use 0's and 1's to represent the *false* and *true* values while x_i's, cx_i's and dx_i's represent distinct variables in the QV_{F_2}, QCV_{F_2} and QDV_{F_2} vectors, respectively, of virtual node F_2. Note that for each (sub-)query of F_2 we introduce a new variable. We use Procedure bottomUp to partially compute the values of the introduced variables, creating a system of Boolean equations.

Observe the following. First, processing at each site invokes Procedure bottomUp for each fragment F_j stored at the site (see Procedure evalQual). For each such fragment, Procedure bottomUp returns a single triplet (QV_{F_j}, QCV_{F_j}, QDV_{F_j}) of vectors that store the (sub-)query values for the root of fragment F_j, for its children and its descendants, respectively. Each site sends the computed triplet(s) to the coordinating site and concludes its computation. Second, in addition to the triplets associated with virtual nodes in a fragment, bottomUp needs only two triplets in total in its process: one for the current node (QV_v, QCV_v, QDV_v) and one for its children (QV_w, QCV_w, QDV_w), rather than assigning a triplet to each node.

Example 3.2. Consider the query from our previous example. At the end of the second phase the following triplets are available to the coordinating site S_0:

— $QV_{F_0} =< 0, 0, 0, 0, 0, dy_5 \vee dz_5 \vee dk_5, dy_5 \vee dz_5 \vee dk_5 >$
   $QCV_{F_0} =< z_1, z_2, z_3, z_4, z_5, z_6, z_7 >$
   $QDV_{F_0} =< 1, dy_2 \vee dz_2 \vee dk_2, dy_3 \vee dz_3 \vee dk_3, 1,$
            $dy_5 \vee dz_5 \vee dk_5, dy_5 \vee dz_5 \vee dk_5 \vee dy_6 \vee dz_6 \vee dk_6,$
            $dy_5 \vee dz_5 \vee dk_5 \vee dy_7 \vee dz_7 \vee dk_7 >$
— $QV_{F_1} =< 0, 0, 0, 0, 0, dx_5, dx_5 >$
   $QCV_{F_1} =< x_1, x_2, x_1 \wedge x_2, x_4, x_4 \wedge x_3, dx_5, dx_5 >$
   $QDV_{F_1} =< dx_1, dx_2, dx_3, dx_4, dx_5, dx_5 \vee dx_6, dx_5 \vee dx_7 >$
— $QV_{F_2} =< 0, 0, 0, 0, 0, 1, 1 >$, $QCV_{F_2} =< 0, 0, 0, 1, 1, 1, 1 >$, $QDV_{F_2} =< 1, 1, 1, 1, 1, 1, 1 >$
— $QV_{F_3} =< 0, 0, 0, 0, 0, 0, 0 >$, $QCV_{F_3} =< 0, 0, 0, 0, 0, 0, 0 >$, $QDV_{F_3} =< 1, 0, 0, 1, 0, 0, 0 >$
— $QV_{F_4} =< 0, 0, 0, 0, 0, 0, 0 >$, $QCV_{F_4} =< 0, 0, 0, 1, 0, 0, 0 >$, $QDV_{F_4} =< 1, 0, 0, 1, 0, 0, 0 >$.

In the triplets, variables x_j and dx_j ($1 \leq j \leq 7$) are used in fragment F_1 to denote the values of virtual node F_2 (see Example 3.1), while dy_i, z_i, dz_i and dk_i are used in fragment F_0 for the values of the virtual nodes F_1, F_3 and F_4, respectively.

Composition of partial answers.. In the third phase of Algorithm ParBoX, the coordinating site uses the computed triplets from all the fragments to evaluate the answer to query q. In a nutshell, the computed triplets form a linear system of Boolean equations. Using the computed vectors and the fragment tree, Procedure evalST (simple and hence omitted) needs a single bottom-up traversal of the fragment tree to solve the quations and find the answer to query q. Note that the vectors of leaf fragments in the fragment tree contain no variables. This is the case for both fragments F_2, F_3 and F_4. During the bottom-up traversal of $\mathcal{S}_T$, evalST uses the Boolean values of the leaf fragments to unify the variables of the vectors that belong to the parent fragments in $\mathcal{S}_T$. The procedure continues in this fashion until it reaches the root of $\mathcal{S}_T$. The answer for query q is the value of $QV_{F_{root}}(q_{last})$, where F_{root} is the root fragment and q_{last} is the last query in the q_L list.

Example 3.3. Consider the fragment tree in Figure 2 and the vectors of the fragments from our previous example. Then, the answer to query $[q]$ is the value of the last query in QV_{F_0}, that is, $QV_{F_0}(q_7) = dy_5 \vee dz_5 \vee dk_5$. A bottom-up evaluation of Procedure evalST uses the F_2-returned vectors to unify dx_5 to 1; the F_1-returned vectors to unify dy_5 to dx_5, that is, to 1; the F_3-returned vectors to unify dz_5 to 0; and the F_4-returned ones to unify dk_5 to 0. Therefore, $QV_{F_0}(q_7) = dy_5 \vee dz_5 \vee dk_5 = 1$ and the query $[q]$ evaluates to *true*.

3.2. Analysis

For the complexity of Algorithm ParBoX, we consider its communication cost as well as the total and parallel computation costs for evaluating a query q on a partitioned and distributed tree T. The total computation cost is the sum of the computation performed at all the sites that participate in the evaluation. In contrast, the parallel computation cost is the time needed for evaluating the query at different sites in parallel. Since a large part of the evaluation is performed in parallel, the parallel computation cost more accurately describes the performance of the algorithm.

We use the following notations: $\mathcal{F}$ denotes the set of all fragments of the original tree T, and $\mathcal{F}_j \subseteq \mathcal{F}$ denotes the subset of fragments of T that are subfragments of fragment F_j. We use card$(\mathcal{X})$ to denote the cardinality of a set $\mathcal{X}$.

Total network traffic. Observe that each site appearing in the fragment tree $\mathcal{S}_T$ of tree T is visited *only once*, when the coordinating site sends the input query q to these sites in the first stage. For each fragment F_j in site S_j the algorithm generates three vectors, each with $O(|q|)$ entries. Each entry may hold a formula computed by Procedure bottomUp, and its size depends on the number of virtual nodes in fragment F_j, that is, card$(\mathcal{F}_j)$, due to the variables introduced by these virtual nodes. In the worst case, the size of the entry is in $O(|\mathcal{F}_j|)$. Thus the communication cost for each fragment F_j is $O(|q|\text{card}(\mathcal{F}_j))$ and the overall communication cost is $O(|q|\Sigma_{j=1}^{\text{card}(\mathcal{F})}\text{card}(\mathcal{F}_j))$, that is, $O(|q|\text{card}(\mathcal{F}))$ (since fragments are disjoint).

Total computation. Site S traverses each fragment F_j assigned to it only once (through Procedure bottomUp). At each node v in a fragment, the procedure takes $O(|q|)$ time and therefore, the cost of the procedure on fragment F_j is $O(|q||F_j|)$. Adding these up for all fragments of tree T, the total amount of computation in the second phase of the algorithm is $O(|q||T|)$. The third phase of the algorithm solves, in linear time, a system of Boolean equations that is of size $O(|q|\text{card}(\mathcal{F}))$. Overall, the total amount of computation of Algorithm ParBoX is $O(|q|(|T| + \text{card}(\mathcal{F})))$.

Parallel computation. The cost of the second phase may differ depending on the level of parallelism. Intuitively, as sets of fragments are assigned to different sites, the cost of the second phase is equal to the computation cost at the site holding the set with the largest aggregated fragment size. We use $|F_{S_i}|$ to denote the sum of the sizes of the fragments in site S_i. Then, the time taken by the second phase is $O(|q|\max_{S_i}(|F_{S_i}|))$ and the parallel computation cost of the algorithm is $O(|q|(\max_{S_i}(|F_{S_i}|) + \text{card}(\mathcal{F})))$.

In any reasonable setting, we expect that the number of fragments to which a tree is decomposed will be small compared to the size of the tree itself, that is, card$(\mathcal{F}) << |T|$. Thus, given a decomposition of a tree T to a set of fragments, Algorithm ParBoX has the desirable property that the communication cost of evaluating a query q over T is *independent of* the size $|T|$ of the tree and depends mainly on the size $|q|$ of the query. Similarly, the total computation cost of Algorithm ParBoX becomes $O(|q||T|)$, comparable to that of the best-known centralized algorithm [Gottlob et al. 2002; Koch 2003]

for evaluating an XPath query q over a tree T. Furthermore, the parallel computation cost depends only on the size of the largest aggregated fragment size assigned to a site.

4. DISTRIBUTED EVALUATION OF DATA-SELECTING XPATH QUERIES

We next present two algorithms, namely Algorithm PaX3 and its optimization Algorithm PaX2, to evaluate data-selecting queries. Algorithms PaX3 and PaX2 provide the following guarantees:

(1) Each site is visited at most three times in PaX3, and at most twice in PaX2, irrespectively of the number of fragments stored in it.
(2) Query processing is performed in parallel, on all the participating sites.
(3) The total computation on all sites is comparable to what of the best-known centralized algorithm.
(4) The total network traffic, in any practical setting, is determined by the size of the query and the size of the query answer rather than the XML tree.

Thus, both algorithms retain all the desirable properties of ParBoX, while evaluating generic data-selecting XPath queries, instead of just Boolean ones.

We will present first PaX3 and then we present its optimized PaX2 version. Algorithm PaX3 is initiated at site S_Q where the query Q is issued. We assume w.l.o.g. that S_Q stores the root fragment of the tree T. As shown in Figure 6, the algorithm has three stages, where each stage corresponds to a single visit of a site holding tree fragments. In turn, each visit makes a single pass of each tree fragment and therefore, overall, algorithm PaX3 makes three passes over the tree T. More specifically:

Stage 1. We first (partially) evaluate the *qualifiers* of query Q, at each node of each fragment using Algorithm ParBoX. At the end of this stage for some nodes we know the actual value of each qualifier, while for other nodes the value for some qualifiers is a *Boolean formula* whose value is yet to be determined. The value of each qualifier (Boolean formula) is known for all nodes by the beginning Stage 2.

Stage 2. The objective of this stage is to (partially) evaluate the *selection part* of query Q. Intuitively, this means that at the end of this stage, for each node of each fragment, we know one of two things: (a) whether or not the node is part of the answer of query Q; or (b) that the node is a candidate answer. Again, candidacy depends on the value of a Boolean formula.

Stage 3. For this latter set of candidate nodes, we need one additional pass during this stage to determine which candidate answer nodes are *true* answer nodes. At the same time, all nodes belonging to the answer of Q are transmitted to site S_Q.

Note that the ParBoX algorithm corresponds only to the first stage of PaX3. The tricky part, namely, finding candidate answer nodes and identifying true answer nodes, is done in Stages 2 and 3. We next present the details of these two stages.

4.1. Selection Path Evaluation

Stage 2 of Algorithm PaX3 is initiated again at site S_Q by having site S_Q notifying each site S_i holding a fragment F_j about the result of Procedure evalFT (Figure 6, line 6), that is, the truth values of triplets $(QV_{F_k}, QCV_{F_k}, QDV_{F_k})$ for each subfragment F_k of F_j (Figure 6, lines 6–8). The triplets received will be used by site S_i to determine the values of qualifiers for all the nodes in F_j. As a next step, site S_Q initiates the partial evaluation of the query selection path by making a remote procedure call (Figure 6, lines 9–10) to all the sites holding at least one tree fragment.

We now examine in detail the partial evaluation of a selection path. Consider a query Q over T and, in particular, the selection path vector SVect(Q) of Q. An efficient

Procedure PaX3

Input: An XPath query Q and a fragmented tree T
Output: The set of nodes ans of Q over T, that is, the answer of Q

    /* Stage 1*/
1. **for** each site S_i in $\mathcal{F}_T$ **do**
2. $exec(S_i, \mathsf{evalQual}, \mathsf{QVect}(Q))$ in parallel;
3.   **for** each fragment F_j stored in S_i **do**
4.   annotate $\mathcal{F}_T$ with $(QV_{F_j}, QCV_{F_j}, QDV_{F_j})$;
5. $\mathsf{evalFT}(\mathcal{F}_T)$;
    /* Stage 2*/
6. **for** each fragment F_j stored in S_i **do**
7.   **for** each subfragment F_k of F_j **do**
8.   send $(QV_{F_k}, QCV_{F_k}, QDV_{F_k})$ to S_i;
9. **for** each site S_i in $\mathcal{F}_T$ **do**
10. $exec(S_i, \mathsf{evalSelQ}, \mathsf{SVect}(Q))$ in parallel;
11.   **for** each fragment F_j stored in S_i **do**
12.   **for** each subfragment F_k of F_j **do**
13.   annotate $\mathcal{F}_T$ with SV_{F_k};
14. $\mathsf{evalFT}(\mathcal{F}_T)$;
    /* Stage 3*/
15. **for** each subfragment F_k of a fragment F_j **do**
16.   send SV_{F_k} to site S_l storing F_k;
17. **for** each site S_i in $\mathcal{F}_T$ **do** $exec(S_i, \mathsf{collectAns})$;
18. receive ans from each site S_i

Fig. 6. PaX3 algorithm executed at site S_Q.

evaluation of $\mathsf{SVect}(Q)$ over T requires a single top-down (depth-first) traversal of T (Figure 7, Procedure $\mathsf{topDown}$). During the traversal, for each node v and for each subquery q_i in $\mathsf{SVect}(Q)$, we decide whether or not v can be reached from the root of the entire tree by following q_i. We store the results for all subqueries of $\mathsf{SVect}(Q)$ in a Boolean vector SV_v associated with node v (Procedure $\mathsf{topDown}$, lines 2–6). This computation often requires to consult the (already computed) values of the $\mathsf{SVect}(Q)$ subqueries at the ancestors of v. To this end, we use a stack to store the values of SV_u, for every node u that is an ancestor of v. At the same time, we make sure that the vector at the top of the stack *summarizes* the information for all vectors in the stack. Specifically, when node v is being processed, the top of the stack holds SV_p, where p is the parent of v. Then, $SV_v(q_i)$ is computed as follows. If q_i is a basic term A, $SV_v(q_i)$ is set *true* if the label of v is A (line 4 of Procedure $\mathsf{topDown}$ and Procedure term of Figure 7). If q_i is q_j/t for some query q_j and a basic term t, then $SV_v(q_i)$ is *true* if both $SV_p(q_j)$ *and* $\mathsf{term}(v, t)$ are *true*. Here, function evalFM (which is similar in spirit to compFm of ParBoX and thus not shown) is used to compute the Boolean formula of $SV_p(q_j) \wedge \mathsf{term}(v, t)$. If q_i is $q_j//$, then $SV_v(q_i)$ is *true* if either $SV_p(q_i)$ *or* $SV_v(q_j)$ is *true* (note that $SV_v(q_j)$ is already computed since it precedes q_i in $\mathsf{SVect}(Q)$). At the end of the computation at node v, we consult the *last entry* in SV_v, denoted by $SV_v(|\mathsf{SVect}(Q)|)$. If this entry is *true* then node v is part of the answer for query Q and is added in the set ans, otherwise it is not. Notice that in terms of space, our vectors are still linear in the size of the query Q. Furthermore, by summarizing the whole stack information at the top of the stack, there are considerable savings in terms of time since we avoid going through the whole stack during each node computation. Also, note that unlike ParBoX, which requires three vectors per node for the evaluation of qualifiers, here we only maintain a *single* vector per node.

Procedure evalSelQ

Input: A vector $\mathsf{SVect}(Q)$ of (sub-)queries
Output: Set returnSet $= \{SV_{F_k}|$ where
$\qquad\qquad F_k$ is a subfragment of a fragment F_j of site $S_i\}$

1. **for** each fragment F_j assigned to S_i **do**
2. $\quad$ cans $:= \emptyset;\quad$ tempSet $:= \mathsf{topDown}(root(F_j), \mathsf{SVect}(Q));$
3. $\quad$ returnSet $:= $ returnSet $\cup$ tempSet;
4. send returnSet to site $S_Q;$

Procedure topDown

Input: A node v and a vector $\mathsf{SVect}(Q)$ of (sub-)queries
Output: Vector SV_v of formulas for node v

1. $\quad$ initStack();
2. $\quad$ **for** each query q_i in $\mathsf{SVect}(Q)$ from left to right **do**
3. $\quad\quad$ **case** q_i **of**
4. $\quad\quad\quad t : SV_v(q_i) := \mathsf{term}(v, t);$
5. $\quad\quad\quad q_j/t\colon SV_v(q_i) := \mathsf{evalFM}(\mathsf{stack}(SV(q_j)), \mathsf{term}(v, t), \wedge);$
6. $\quad\quad\quad q_j//\colon SV_v(q_i) := \mathsf{evalFM}(\mathsf{stack}(SV(q_i)), SV_v(q_j), \vee);$
7. $\quad\quad SV_v(q_i) := \mathsf{evalFM}(SV_v(q_i), \mathsf{assocQual}(QV_v), \wedge);$
8. $\quad$ pushStack(v, SV_v)
9. $\quad$ **if** $SV_v(|\mathsf{SVect}(Q)|) = true$ **then** ans $:= $ ans $\cup\, v;$
10. $\quad$ **elseif** $SV_v(|\mathsf{SVect}(Q)|)$ is a formula **then**
11. $\quad\quad$ cans $:= $ cans $\cup\, (v, SV_v(|\mathsf{SVect}(Q)|));$
12. $\quad$ **if** v is a virtual node **then** returnSet $:= $ returnSet $\cup\, SV_v$
13. $\quad$ **for** each child w of v **do** $SV_w := \mathsf{topDown}(w, \mathsf{SVect}(Q));$
14. $\quad$ popStack();

Procedure term

Input: A node v and a terminal symbol t in the normal of q
Output: The truth value of evaluating t on v

1. **case** t **of**
2. $\quad\quad \epsilon\colon$ **return** $true;$
3. $\quad\quad *\colon$ **return** $true;$
4. $\quad\quad label() = l\colon$ **return** $compareString(label(), l);$
5. $\quad\quad text() = str\colon$ **return** $compareString(text(), str);$

Procedure collectAns

Input: Vector SV_{F_j} of a fragment F_j of site S_i
Output: The set of nodes ans of q over F_j

1. **for** each pair $(v, SV_v(|\mathsf{SVect}(Q)|)$ in cans **do**
2. $\quad$ **if** $\mathsf{unify}(SV_v(|\mathsf{SVect}(Q)|)) = true$ **then** ans $:= $ ans $\cup\, \{v\};$
3. return ans

Fig. 7. PaX3 algorithm executed at participating sites.

There are two more things to consider. First, we need to consider qualifiers. Recall that qualifiers and selection paths are evaluated independently by Algorithm PaX3. We maintain the relationship between the selection $\mathsf{SVect}(Q)$ and qualifier part $\mathsf{QVect}(Q)$ of query Q through Procedure $\mathsf{assocQual}$ (Procedure topDown, line 7). The procedure returns for each entry of $\mathsf{SVect}(Q)$ the qualifier entry in $\mathsf{QVect}(Q)$ whose truth value needs to be checked. To determine this truth value, we only need to instantiate the variables in QV_v with the corresponding truth values in QV_{F_k}, for

subfragment F_k. These latter truth values have already been computed during Stage 1 and are known at the beginning of this stage.

Example 4.1. For simplicity, ignore for the moment the fragmentation of the tree in Figure 1 and assume that we evaluate SVect(Q) from Example 2.1 over the three *client*s of the tree. Then, the following are the SVect(Q) vectors computed for some of the nodes:

Vectors	leftmost *client*	middle *client*	rightmost *client*
SV_{client}	$< 1, 0, 0 >$	$< 1, 0, 0 >$	$< 0, 0, 0 >$
SV_{broker}	$< 0, 1, 0 >$	$< 0, 1, 0 >$	$< 0, 0, 0 >$
SV_{name}	$< 0, 0, 1 >$	$< 0, 0, 1 >$	$< 0, 0, 0 >.$

For the two leftmost clients all vectors are identical since for both clients the qualifiers on *client* and *broker* evaluate to true (from Stage 1). Hence the *name* nodes of *broker* are answers to the query, verified by the fact that $SV_{name}(|\mathsf{SVect}(Q)|)$ is true. For the rightmost client, although there exists a *client/broker/name* path, the vector entries are all false since all qualifiers evaluate to false.

The second issue concerns partitioning. At the beginning of the top-down traversal, given the root node r of a fragment F_j, we do not know the SVect(Q) vector which summarizes the ancestors of r (located in some other fragment). Similar to the evaluation of qualifiers, we address this issue by introducing *Boolean variables*, one for each value of the unknown SVect(Q) vector. We initialize the stack used in the traversal to include the vector with the variables (Procedure topDown, line 1). An immediate effect from the introduction of variables is that for some nodes, say node v, the last entry in vector SV_v might be a Boolean formula. Since we are not sure about whether or not node v is an answer to Q, we add v to the set of candidate answers of Q (Procedure topDown, lines 10–11). The third stage of algorithm PaX3 will determine which of the candidate answers is an actual answer to Q.

Similar to Stage 1, Stage 2 concludes by having each site S_i returning to site S_Q a set of SVect(Q) vectors, namely, returnSet, one vector for each subfragment (virtual node) F_k of a fragment F_j of site S_i. Note that returnSet consists of at most k Boolean vectors, where k is the number of virtual nodes in the fragment. Neither ans nor cans is sent to S_Q. At site S_Q, Procedure evalFT unifies the variables in the received vectors, through a single top-down traversal of $\mathcal{F}_T$.

Example 4.2. After Procedure topDown concludes in fragment F_1, the following SVect(Q) vectors are computed for the nodes in F_1.

$$SV_{init} =< z_1, z_2, z_3 >, \qquad SV_{broker} =< 0, z_1, 0 >, \qquad SV_{name} =< 0, 0, z_1 > .$$

Vector SV_{init} is inserted into the stack in Procedure topDown (line 1). This is because we are not sure during the parallel processing of fragments what path precedes node *broker*. In this particular case, we are interested in whether the parent node of *broker*, which is stored in fragment F_0, is *client* (variable z_1). Even if we know that the parent node of *broker* *must* be *client*, we are not certain whether there are any qualifiers in the parent node or whether any such qualifiers evaluate to true or false. One of the advantages of partial evaluation is that query processing proceeds, even in the presence of uncertainty, and information about qualifiers and selection paths is kept *local* to each fragment rather than being sent to the coordinator site S_Q. This results in, as we will prove in Section 4.3, minimum network traffic while computation costs remain optimal.

The uncertainty of what precedes node *broker* is propagated in both SVect(Q) vectors SV_{broker} and SV_{name} through Boolean variable z_1. Note that node *name* is a

candidate answer due to the last entry in SV_{name}. After Procedure topDown concludes in all fragments, Procedure evalFT uses vector $SV_{client} = <1, 0, 0>$ from fragment F_0 to unify vector $SV_{init} = <z_1, z_2, z_3>$ from fragment F_1. Variable z_1 is unified to true and thus node *name* is an answer to Q.

4.2. Retrieving Query Answers

The last stage of Algorithm PaX3 is initiated at site S_Q by having site S_Q notifying each site S_l holding a fragment F_k about the result of Procedure evalFT (Figuire 6, line 15), that is, the truth values of vector SV_{F_k}. Although vector SV_{F_k} was sent to S_Q from site S_i holding the parent fragment F_j of F_k, the vector is sent to S_l in which F_k is stored, instead of S_i. The received vectors are used by each site to decide which candidate answers in cans are real answers to Q (Figure 7, Procedure collectAns). Referring to our last example, after site S_Q determines that variable z_1 unifies to true, it sends this information to fragment F_1. Fragment F_1 decides, in turn, that node *name* is an answer to query Q and thus it sends this node back to site S_Q.

4.3. Analysis

To draw comparisons with ParBoX, for the analysis of PaX3 we again consider the communication cost of the algorithm as well as its total and parallel computation costs. Recall that the total computation cost is the sum of the computation performed at all the sites while the parallel computation cost is the time needed for evaluating the query at different sites in parallel.

Communication cost. As shown in the analysis of ParBoX, the communication cost for the first stage is $O(|Q| \ |\mathcal{F}_T|)$, that is, communication is independent of the initial tree T and it only depends on the size of Q. The second stage of PaX3 also has cost $O(|Q| \ |\mathcal{F}_T|)$, while the last stage has cost $O((|Q| \ |\mathcal{F}_T|) + |ans|)$. Therefore, the total communication cost of PaX3 is $O((|Q| \ |\mathcal{F}_T|) + |ans|)$.

When we execute a query Q in a distributed environment with $|\mathcal{F}_T|$ sites, it is obvious that we are willing to pay at least the cost of transmitting our query over the various sites (cost $O(|Q| \ |\mathcal{F}_T|)$). In addition, one cannot avoid the cost of retrieving the actual answers to our query (cost $O(|ans|)$). In this sense, a communication cost $O((|Q| \ |\mathcal{F}_T|) + |ans|)$ is *optimal*.

Total computation cost. At each stage, each fragment F_j is traversed only once. During the traversal, at each node v of F_j at most $O(|Q|)$ operations are performed (one operation per vector entry). Therefore, at each stage, the total computation for each fragment is $O(|Q| \ |F_j|)$. At the end, in each stage and overall, the total computation for all fragments is $O(|Q| \ |T|)$, which coincides with the cost of executing query Q over T in a central site [Gottlob et al. 2002]. Therefore, the distribution of computation does not incur extra computation costs.

Parallel computation cost. As more than one fragment can be assigned to a site, we use $|F_{S_i}|$ to denote the cumulative size of the fragments in site S_i. As computation is performed in parallel at all sites, the parallel computation cost at each stage is determined by the site holding the largest cumulative fragment. That is, the parallel computation cost for the first two stages and overall is $O(|Q| \ \max_{S_i} |F_{S_i}|)$.

Correctness. One can verify, by induction on the structure of $\mathcal{X}$ queries Q, that algorithm PaX3 computes the correct answer $Q(T)$ on any XML tree T no matter how T is fragmented and distributed.

```
Procedure PaX2
Input: An XPath query Q and a fragmented tree T
Output: The set of nodes ans of Q over T
      /* Stage 1 */
1.   for each fragment F_j stored in S_i do
2.       exec(S_i, F_j, evalXPath, SVect(Q));
3.   for each fragment F_j stored in S_i do
4.       annotate F_T with (QV_{F_j}, QCV_{F_j}, QDV_{F_j});
5.       for each subfragment F_k of F_j do
6.           annotate F_T with SV_{F_k};
7.   evalFT(F_T);
      /* State 2 */
8.   for each fragment F_j stored in S_i do
9.       for each subfragment F_k of F_j do
10.          send (QV_{F_k}, QCV_{F_k}, QDV_{F_k}) to S_i;
11.          send SV_{F_k} to site S_l storing F_k;
12.  for each site S_i in F_T do exec(S_i, collectAns);
13.  receive ans;
```

Fig. 8. The PaX2 Algorithm.

Summary. With the exception of the necessary $O(|\mathsf{ans}|)$ cost incurred by transmitting query answers and at most three visits per site, the costs of PaX3 coincide with those of ParBoX. In short, we have proposed an algorithm to evaluate a larger, and more useful, fragment of queries than those considered by ParBoX yet we provided comparable performance guarantees. Moreover, we guarantee minimum data transmission since the only data transmitted by PaX3 are the actual query answers.

4.4. Improving Algorithm Pax3

We next present Algorithm PaX2, which has two stages and needs only two visits per site, one less than PaX3.

The main idea behind algorithm PaX2 is to combine the first two stages of algorithm PaX3, that is, the evaluation of qualifiers and selection paths, into a single stage. As shown in Figure 8, the algorithm starts with site S_Q, making a remote procedure call to all the sites holding fragments of T (lines 1–2). At each such site, Procedure evalXPath *combines* the partial evaluation of selection paths with that of qualifiers, over a fragment F_j. The procedure performs a top-down (depth-first) traversal of fragment F_j. At each node v of F_j, two types of computation are performed: a preorder computation and a post-order computation. The preorder computation at v essentially performs the computation of Procedure topDown in Figure 7. One important difference is that unlike Procedure topDown which assumes that qualifiers have already been computed (Figure 7, line 7), here we need to introduce variables for the values of the yet undetermined qualifiers.

Example 4.3. Consider the query of Example 2.1. The preorder computation of the query over the leftmost *client* node of fragment F_0 (shown in Figure 3(a)) results in the SVect(Q) vector $SV_{client} =< qz_1, 0, 0 >$. Variable qz_1 indicates that although the node label is *client*, the qualifier for the node is yet to be determined. Contrast this with Example 4.1 where, for the same node, PaX3 results in vector $< 1, 0, 0 >$ since the value of qualifiers has already been computed by Stage 1 of PaX3.

For a more complex example, consider the preorder computation over fragment F_1. The computation results in $SV_{broker} =< 0, z_1 \wedge qz_2, 0 >$. Here variable z_1 is due to

the initialization of the vector stack (see Example 4.2), while variable qz_2 is due to the qualifier for the node that is still undetermined. In terms of the *name* node in F_1, $SV_{name} =< 0, 0, z_1 \wedge qz_2 >$

The postorder computation at a node v starts once every node in the subtree rooted at v is visited (always within a fragment). At that point, the qualifiers have been computed for all the nodes in the subtree, through a procedure similar to ParBoX. Using the qualifier values at node v, we can unify some of the variables introduced during the preorder computation.

Example 4.4. Going back to the query from Example 2.1, after we traverse the subtree rooted at the leftmost *client* node of F_0, the qualifiers for the subtree rooted at *client* are as follows:

$$QV_{name} = \quad <0, \quad 0, \quad 0, \quad 0, \quad 1, \quad 0, \quad 0, \quad 0, \quad 0>$$
$$QV_{country} = <1, \quad 0, \quad 1, \quad 0, \quad 0, \quad 0, \quad 0, \quad 0, \quad 0>$$
$$QV_{F_1} = \quad <x_1, \ x_2, \ x_3, \ x_4, \ x_5, \ x_6, \ x_7, \ x_8, \ x_9>$$
$$QV_{client} = \quad <0, \quad 0, \quad 0, \quad 1, \quad 0, \quad 0, \quad 0, \quad 0, \quad x_8>$$

The *client* node is associated only with the first qualifier (entry q_4 of QV_{client}). Thus we can unify variable qz_1 in $SV_{client} =< qz_1, 0, 0 >$ (see the last example) to true, yielding vector $< 1, 0, 0 >$, the same vector that PaX3 computes but only after two passes.

For fragment F_1, the qualifiers for the subtree rooted at *broker* are shown here:

$$QV_{broker} = \quad <0, \ 0, \ 0, \ y_3, \ 0, \ 0, \ 0, \ 0, \ y_8>$$
$$QV_{F_2} = \quad < y_1, y_2, y_3, \ y_4, \ y_5, \ y_6, y_7, y_8, y_9 >$$

Node *broker* is associated only with the second qualifier (entry q_9 of QV_{broker}). Thus we can unify variable qz_2 in $SV_{broker} =< 0, z_1 \wedge qz_2, 0 >$ (see the last example) to y_8, yielding $< 0, z_1 \wedge y_8, 0 >$, while SV_{name} is now $< 0, 0, z_1 \wedge y_8 >$. The values for both variables z_1 and y_8 will be determined in the next stage of PaX2.

Stage 1 concludes by having each site S_i returning to site S_Q a set of SVect(Q) and QVect(Q) vectors, one SVect(Q) vector for each virtual node F_k of a fragment F_j of site S_i, and one QVect(Q) vector for each fragment F_j of site S_i. Then at S_Q Procedure evalFT unifies the variables in the received vectors. Stage 2 of PaX2 is similar to Stage 3 of PaX3. The unified vectors are sent from S_Q to the appropriate sites, and the sites return to S_Q the query answers.

Example 4.5. Continuing our last example, site S_1 holding fragment F_1 receives the following vectors from S_Q:

$$SV_{init} =< 1, 0, 0 >, \qquad QV_{F_2} =< 0, 0, 0, 0, 0, 0, 0, 1, 0 >$$

With these vectors, site S_1 unifies both variables z_1 (entry q_1 in SV_{init}) and y_8 (entry q_8 in QV_{F_2}) to true. Then the vector for node *broker* becomes $< 0, 1, 0 >$ and the vector for node *name* becomes $< 0, 0, 1 >$. Therefore, node *name* is in the answer of Q.

Analysis. While the worst-case complexity of algorithms PaX2 and PaX3 is the same, PaX2 requires *one less* visit per fragment. Indeed, our experimental results in Section 7 demonstrate that PaX2 outperforms PaX3.
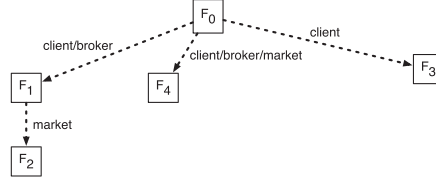
Fig. 9. XPath-annotated fragment tree.

4.5. Optimization via Annotations

We present an optimization method that identifies fragments that do not contain any nodes in the query answer. The optimization technique is used by both PaX3 and PaX2 to rule out identified fragments from any processing.

To do this, we require that each edge (F_j, F_k) of the fragment tree $\mathcal{F}_T$ of T is annotated with a simple XPath expression describing the path in T connecting the root of fragment F_j with the root of fragment F_k. As an example, Figure 9 shows the XPath-annotated fragment tree from our motivating example. The (F_0, F_4) edge is annotated with *client/broker/market* since the root of fragment F_4 is reachable from the *clientele* root node through this expression. The additional XPath-annotation requirement imposes negligible space overhead for the fragment tree $\mathcal{F}_T$.

To see how XPath-annotations can help during query evaluation, consider a query Q and a top-down evaluation of the selection path of Q. Both algorithms PaX3 (in its Stage 2) and PaX2 (in its Stage 1) adopts this strategy. We first use a top-down evaluation of the selection path of Q over the XPath-annotated fragment tree. Performing such an evaluation results in a set of nodes that correspond to fragments. Each returned fragment potentially contains actual tree nodes that are in the answer of Q. Our objective is to evaluate PaX3 or PaX2 only on these fragments and skip fragments that we know contain no nodes relevant to the answering of Q.

Example 4.6. Consider a simple query *client/name* over tree T, which returns the names of all clients. Evaluating this query over the fragment tree of Figure 9 finds fragments F_0 and F_3. Fragment F_0 is considered since the procedure cannot determine with certainty whether the fragment contains or not any paths satisfying the query. Fragment F_3 is considered since the procedure knows that a client subtree is included in F_3, although it is not certain whether a *name* node is also included. On the other hand, the procedure determines with certainty that fragments F_1, F_2 and F_4 should not be considered. Fragment F_1 is ruled out since the path *client/broker* between fragments F_0 and F_1 does not satisfy the query. Similarly, the path *client/broker/marker* from F_0 to F_2 and F_4 does not satisfy the query and therefore, fragments F_2 and F_4 are both ruled out.

XPath-annotations are used before the beginning of Stage 2 of PaX3 (resp. before Stage 1 of PaX2) to identify fragments that are relevant to a query. Apart from ruling out irrelevant fragments, XPath-annotations can also be used to reduce the number of passes of PaX3 and PaX2. More specifically, if the input query Q has no qualifiers then we can use XPath-annotations to skip the last step of both algorithm PaX3 and PaX2. Intuitively, through the XPath-annotations we can guarantee that any candidate answers identified by Stage 2 of PaX3 (resp. Stage 1 of PaX2) are real answers to the query and can be sent back to site S_Q. Recall from Section 4.1 that without XPath-annotations, for each fragment, we need to initialize the stack in Procedure topDown with variables, since we have no information about the ancestor nodes of each fragment root. XPath annotations encapsulate precisely the information about the ancestors of

a fragment root and they can be used to initialize the stack in Procedure topDown with concrete Boolean values, instead of variables. Thus every answer to query Q can be identified with certainty.

As will be seen in Section 8, XPath annotation has been explored [Marian and Siméon 2003; Zhang et al. 2009] for query processing. The previous discussion demonstrates that XPath annotation is also effective in partial evaluation, as will be experimentally verified in Section 7.

5. A MAP-REDUCE ALGORITHM FOR EVALUATING BOOLEAN QUERIES

MapReduce [Dean and Ghemawat 2004] is a software framework for processing huge datasets on certain distributable problems using a large number of computers. It allows distributed and parallel processing of map and reduction operations, and hence, can be applied to huge datasets, and facilitate recovering from partial failure of servers or storage during the operations. It has found a wide range of applications in industry.

Partial evaluation can be naturally expressed in the MapReduce model. To demonstrate this we next present Algorithm MRParBoX (Map Reduce based ParBoX), a MapReduce algorithm for evaluating XPath queries based on partial evaluation. To simplify the discussion we focus on Boolean XPath queries.

5.1. The MapReduce Model

MapReduce [Dean and Ghemawat 2004] is a model for data-intensive parallel computation in shared-nothing clusters. In MapReduce, data are modeled as <*key, value*> pairs, partitioned across the nodes of a cluster and stored in a DFS (Distributed File System). The computation is expressed using two user-defined functions:

$$\text{map} <\text{k1,v1}> \rightarrow \text{list}(<\text{k2,v2}>); \quad \text{reduce} <\text{k2,list(v2)}> \rightarrow \text{list}<\text{k3,v3}>.$$

The computation is distributed across the nodes of the cluster. (a) A designated coordinator first partitions the input data into *input splits*, and each of the splits is assigned to a machine as a *map* task. (b) The map function is applied to different input splits in parallel, producing list(<k2,v2>) of intermediate <*key, value*> pairs. (c) The <*key, value*> pairs produced in the map phase are hash-partitioned based on the *key*, yielding partitions <k2,list(v2)>. Each of the partitions is assigned to a machine as a *reduce* task. (d) The reduce function is invoked for each <k2,list(v2)>, producing <*key, value*> pairs that are written to a distributed file in the DFS.

5.2. The MRParBoX Algorithm

We now present Algorithm MRParBoX, shown in Figure 10. It consists of three procedures, configMRParBoX, mapParBoX and reduceParBoX, described as follows.

Procedure configMRParBoX is invoked before map or reduce functions are called. It first loads input $\mathcal{X}_{\text{BL}}$ query q and parses q into QVect (q) (lines 1–2). It then loads fragment tree $\mathcal{F}_T$ (line 3). All subsequent map tasks need to utilize q and $\mathcal{F}_T$.

Function mapParBoX takes as input a <*key, value*> pair, where *key* is the id of a fragment F of the input XML tree T, and *value* is the XML content of the fragment F. To accommodate arbitrary fragmentation strategies, we assume that the XML data are already partitioned, as before, but remark that the algorithm can be readily extended to process data that are not yet partitioned. The function conducts partial evaluation on fragment F, along the same lines as ParBoX. Similar to ParBoX, for virtual leaves in F that link to other fragments, mapParBoX uses variables to represent them and leaves it to the reduce phase to find the true value of those variables.

More specifically, when processing a pair <k1, v1>, mapParBoX first reads v1 and calls buildFragment to construct XML tree F_{k1} (line 1), where buildFragment can be

```
Procedure configMRParBoX

1. Load input $\mathcal{X}_{BL}$ query $q$;
2. $q_L := \mathsf{QVect}(q)$;
3. Load the fragment tree $\mathcal{F}_T$;

Procedure mapParBoX

Input: <k1,v1>, k1: fragment ID, v1: the content of fragment k1.
Output: $\mathcal{L}_2$: list of <k2, v2>, k2: intermediate id, v2: Boolean formula.

1. $F_{k1} := \mathsf{buildFragment}(v1)$;
2. $(QV_{F_{k1}}, QCV_{F_{k1}}, QDV_{F_{k1}}) := \mathsf{bottomUp}(F_{k1}, q_L)$;
3. $\mathcal{L}_2 := \mathcal{L}_2 \cup <1, [k1, QV_{F_k}, QCV_{F_{k1}}, QDV_{F_{k1}}]>$;
4. return $\mathcal{L}_2$;

Procedure reduceParBoX

Input: <k2, list(v2)>.
Output: <unused, the Boolean value ans of $q$>.

1. for each pair <k2, v2> in list(v2) do
2.     extract variables vectors from list(v2);
3. $ans :=$ the answer to input $\mathcal{X}_{BL}$ query $q$ composed by procedure evalST;
4. return <0, $ans$>;
```
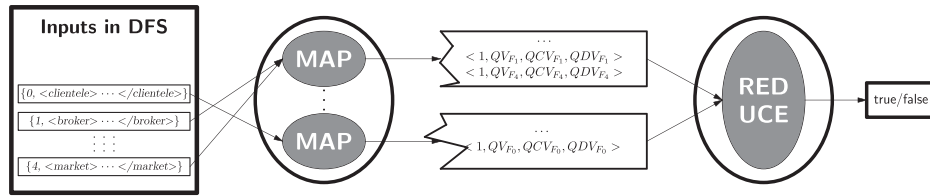
Fig. 10. Algorithm MRParBoX.

Fig. 11. The sketch of MapReduce.

implemented easily by any standard XML parser. It then invokes procedure bottomUp to evaluate q_L over F_{k1} (line 2), and the results are attached to the output list $\mathcal{L}_2$ (line 3). It assigns an intermediate constant key 1 to the Boolean formula computed (line 3).

Function reduceParBoX takes as input <k2, list(v2)>, which is obtained by grouping the output of mapParBoX by *key*. For each pair <k2, v2>, reduceParBoX first extracts the Boolean formula from v2 (lines 1–2). It then invokes procedure evalST (see Section 3) to compose partial answers and get the final answer to q (line 3).

Example 5.1. Continuing with Examples 3.1 and 3.2, consider the same $\mathcal{X}_{BL}$ query [q] given in Example 3.1 posed on the XML tree T shown in Figure 1. As shown in Figure 11 (not including preprocessing by configMRParBoX), Algorithm MRParBoX evaluates [q] over T as follows. Suppose that T has been partitioned, and the fragments have been loaded into DFS and represented as <*key, value*> pairs. First, the fragments are grouped and are sent to map nodes in a cluster of machines. Upon receiving the fragments assigned to it, each map node parses them into <*key, value*> pairs and invokes procedure mapParBoX for each pair. As shown in Figure 11, the output pairs of the mapParBoX function, which consist of vectors QVs, QCVs and QDVs (see Examples 3.1 and 3.2), are grouped by *key*, and are sent to a reduce node as input. The reduce node invokes procedure reduceParBoX to conduct partial answer composition and find the final answer to [q] in T.

```
┌─────────────────────────────────────────────────────────────┐
│ Procedure BT-CPax                                           │
│                                                             │
│ Input: An XPath query Q and a fragmented tree T             │
│ Output: The set ans of nodes of Q over T                    │
│                                                             │
│     /* Stage 1 */                                           │
│ 1.  for each fragment F_j in T, traversed bottom-up, do     │
│ 2.      evalQual(F_j);                                       │
│     /* Stage 2 */                                           │
│ 3.  for each fragment F_j in T, traversed top-down, do      │
│ 4.      evaluate selection path of Q on fragment F_j using  │
│         an adapted Procedure topDown and return results;    │
└─────────────────────────────────────────────────────────────┘
```

Fig. 12.   The BT-CPax Algorithm.

6. XPATH EVALUATION ON LARGE XML DOCUMENTS

Partial evaluation also finds applications in the centralized setting. The idea is to first partition an XML file into smaller fragments that can easily fit into memory, and then extend the partial evaluation techniques to evaluate the queries on each fragment and compute the final answer by assembling partial results. As remarked earlier, the benefits are twofold. First, this no longer requires to load the document into an XML processing system. Second, it overcomes the limitations of those XML engines that have to load (part of) an XML tree into memory to evaluate an query, and no longer work when the data loaded exceeds the capacity of the memory.

We next present two centralized algorithms for evaluating XPath query on large XML documents. Consider an XPath query Q and an XML tree T, which, without loss of generality, is partitioned into n fragments $\mathcal{F}_T = \{F_1, F_2, ..., F_n\}$. There have already been several proposals for partitioning an XML tree [Bordawekar and Shmueli 2004; Kanne and Moerkotte 2006a; Kundu and Misra 1977; Lukes 1974]. Hence we shall simply adopt one of the partitioning strategies, and focus on how to extend the partial evaluation techniques to the centralized setting.

Similar to their distributed counterparts, the centralized algorithms need to first evaluate the qualifiers in the query Q, and then process the selection path of Q when the truth values of the qualifiers are available at each node of the XML tree T. In contrast to the distributed setting, where fragments are distributed across different sites, in the centralized setting all the fragments are in a single site. Thus, when developing the evaluation algorithms, we have the flexibility in terms of the fragments processing order, both for the qualifier and the selection path evaluation.

Our centralized algorithms adopt different evaluation ordering for fragments. One of our algorithms, referred to as BT-CPaX, evaluates qualifiers bottom-up and the selection path top-down. The other, TT-CPaX, evaluates both qualifiers and the selection path top-down. We remark that the optimization techniques given earlier can be readily adapted to the centralized setting.

6.1. Centralized Evaluation Algorithm BT-CPax

Algorithm BT-CPax is shown in Figure 12. Along the same lines as its distributed counterparts, it consists of two stages, where each stage corresponds to one visit to all fragments, described as follows.

Stage 1. This stage is to evaluate the *qualifiers* of the query Q, such that truth value of each qualifier will be known for all nodes by the end of this stage. Obviously, if a query has no qualifiers, this stage can be skipped.

We traverse the fragments bottom-up. This simplifies the evaluation of qualifiers: the truth value of each qualifier at a node can be determined without using any

```
┌─────────────────────────────────────────────────────┐
│  Procedure TT-CPax                                   │
│  Input: An XPath query Q and a fragmented tree T     │
│  Output: The set ans of nodes of Q over T            │
│        /* Stage 1 */                                 │
│  1.  for each fragment Fj in T, traversed top-down, do │
│  2.      evaluate subqualifiers and selection path of Q on │
│          fragment Fj using a combination of preorder │
│          and postorder computation;                  │
│        /* Stage 2 */                                 │
│  3.  for each fragment Fj in T, traversed top-down, do │
│  4.      compute the true value of Boolean expressions │
│          and return results;                         │
└─────────────────────────────────────────────────────┘
```

Fig. 13.   The TT-CPax Algorithm.

Boolean variables or formulas. This is because all the children fragments of a fragment F_j are traversed before F_j, that is, the truth value of qualifiers at a virtual node (representing a child fragment of F_j) in the fragment F_j can be determined before evaluating the qualifiers on the nodes in F_j. Within each fragment, we use an extension of the ParBoX algorithm of Figure 5 to evaluate the qualifiers.

Example 6.1. Consider the query Q of Example 2.1. Suppose that the fragment tree is given in Figure 9. To evaluate Q, the first stage of BT-CPax visits the fragments in the order $F_2 \rightarrow F_1 \rightarrow F_4 \rightarrow F_3 \rightarrow F_0$ to evaluate the *qualifiers* in Q at each node. It first visits fragment F_2 and evaluate subqualifiers at each node. It then visits fragment F_1 that includes the virtual node for F_2. We do not need to introduce variables for the subqualifiers at the virtual node since we have already traversed F_2, that is, the values of vector QVect for F_2 are already known.

Stage 2. This stage aims to evaluate the *selection part* of query Q, such that at the end of this stage, for each node in each fragment, we know whether the node is part of the answer to query Q. Unlike Stage 1, here we traverse the fragments top-down. As expected, again we do not need to introduce Boolean variables or formulas to represent the unknown values of subqueries of the selection path.

Example 6.2. Consider Example 6.1. Stage 2 of Algorithm BT-CPax visits the fragments in the order $F_0 \rightarrow F_3 \rightarrow F_4 \rightarrow F_1 \rightarrow F_2$ to evaluate the *selection* part of Q. It first traverses fragment F_0. When it comes to fragment F_1, the initial value SV_{init} has been determined to be $< 1, 0, 0 >$ as F_0 has already been processed.

6.2. Centralized Evaluation Algorithm TT-CPax

We present algorithm TT-CPax in Figure 13. It consists of two stages, as follows.

Stage 1. In this stage, we evaluate both the *qualifier* and *selection* path of query Q in one traversal of the fragments of the XML tree. The truth value of each qualifier, and the truth value of each selection subqueries for each node will be known or be represented by a Boolean formula by the end of this stage.

We visit the fragments top-down. For each fragment F_j, similar to Algorithm PaX2, we combine the evaluation of qualifiers and that of selection paths by performing a top-down (depth-first) traversal of nodes. At each node v of F_j, two types of computation are performed: a preorder computation for selection path evaluation and a postorder computation for qualifier evaluation. Compared to Algorithm BT-CPax, there are clear benefits by combining the steps. However, these benefits come at a price: we need to

use Boolean variables for the values of subqualifiers and subqueries that cannot be determined at the time of node-processing.

Yet, the combination of qualifier and selection path evaluation opens the way for optimization strategies that are not possible for BT-CPax. Indeed, by separating the two types of evaluation, algorithm BT-CPax must evaluate all the subqualifiers at each node. However, (sub-)qualifiers are associated with only certain subqueries or certain nodes in an XML tree. For example, in Example 2.1, subqualifier $q_4 = */\epsilon[q_3]$ is associated with selecting subquery $q_1 = client$, but not with the others, and subqualifier $q_9 = */\epsilon[q_8]$ is associated with selecting subquery $q_2 = q_1/broker$. Also, subqualifier $q_2 = [text()="US"]$ is only associated with *country* nodes in an XML tree. In BT-CPax the two types of evaluation are performed in different stages. When evaluating qualifiers, the algorithm is unaware of which subquery in the selection path the current node will map to (if any). Thus BT-CPax must evaluate all subqualifiers at each node. In contrast, TT-CPax does not have to do this.

When it comes to implementation, the association between subqualifiers and selecting subqueries can be formally modeled in terms of filtering nondeterministic finite state automata [Fan et al. 2007]. If the truth value of a subquery at a node is false, then there is no need to evaluate the subqualifiers associated with the subquery at the node. Algorithm TT-CPax performs the evaluation of subqueries of selection path in a preorder traversal, and thus we know the truth value or the Boolean formula of subqueries at a node when we evaluate the subqualifiers in a postorder traversal. If the truth value of a subquery at a node is false, we simply discard the subqualifiers associated with the subquery at the node.

Example 6.3. Consider the setting in Example 6.1. Algorithm TT-CPax visits the fragments in the order $F_0 \rightarrow F_3 \rightarrow F_4 \rightarrow F_1 \rightarrow F_2$. In the first stage of TT-CPax, TT-CPax combines the preorder computation for selection path evaluation and the postorder computation for qualifier. Consider the node *client* in fragment F_3. We know that the node is associated only with the first qualifier (entry q_4 of QV_{client}) and thus at the postorder computation, we do not need to evaluate the remaining qualifiers of QV_{client} in the postorder computation (and similarly for the *broker* node and its associated second qualifier in entry q_9 of QV_{client}).

Stage 2. In the second stage of TT-CPax, we compute the truth values of Boolean formulas of subqualifiers and subqueries to determine which candidate nodes are *true* answer nodes. We process the fragments in a top-down order to evaluate the Boolean formulas in a fragment.

7. EXPERIMENTAL STUDY

We provide an experimental study of our algorithms for evaluating XPath queries in the distributed setting (Section 7.1), the MapReduce algorithm (Section 7.2), and of our algorithms for evaluating queries in the centralized setting (Section 7.3).

7.1. Experiments for Distributed Query Evaluation Algorithms

Experimental Setting. Our datasets consist of XML trees in which the root node is simply called a *"site,"* and each child node of the root node is called an XMark [Schmidt et al. 2002] *"site."* We generated multiple XMark *"sites."* In each experiment we assigned (fragments of) XMark *"sites"* to different machines.

Figure 14 shows a sample of the executed queries over our fragmented tree. The choice of presented queries will become clear shortly. Note that we only report results on data selecting XPath queries since they encompass Boolean queries.

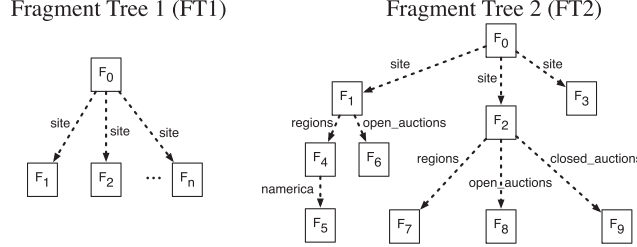| $Q1$	`/site/people/person`
$Q2$	`/site/open_auctions//annotation`
$Q3$	`/site/people/person[profile/age>20 and address/country=‘‘UNITED STATES’’]/creditcard`
$Q4$	`//people/person[profile/age>20 and address/country=‘‘UNITED STATES’’]/creditcard`

Fig. 14.   Sample queries.

Fig. 15.   (Annotated) fragment trees used.

We used ten machines running Redhat Linux 9 (fragment sites), distributed over a local LAN. Each machine has a 3GHz CPU and 1GB of memory. In all experiments, reported times are averaged over multiple runs. For each reported time, computation time dominates over communication time, that is, the time it takes to send query answers to the query site is negligible compared to the time to compute these answer by running PaX3 or PaX2. For consistency, algorithm PaX3 is always plotted using solid lines, while PaX2 is plotted with dotted lines. During the evaluation of an algorithm, if no XPath annotations (NA) are used then the line is plotted using a box symbol, and otherwise it is plotted using a black diamond.

Utility of Fragmentation and Efficiency. The objective of this set of experiments is twofold. First, we want to illustrate the benefits of fragmentation. Second, we want to verify the efficiency and scalability (in the number of fragments) of PaX3 and PaX2, in the presence or in the absence of fragment tree XPath annotations.

We consider a simple fragment tree like FT1, shown to the left of Figure 15. Our conclusions carry over to more complex fragment trees (with the same number of fragments) since in both PaX3 and PaX2, irrespectively of the structure of the fragment tree and the presence of XPath annotations, a site holding a fragment at any level of the tree communicates directly with site S_Q. Each fragment in FT1 corresponds to an XMark *"site"* and is assigned to a different machine. Throughout this experiment, the cumulative size of all fragments in FT1 is constant and equal to approx. 100MB. In more detail, in the first iteration of the experiment we consider a single fragment F_0 of size 100MB, then iteration two considers two fragments F_0 and F_1 of 50MB each and, in general, in iteration j we consider j fragments each of size $(100/j)$MB. For this experiment, we focus on two queries, one without qualifiers ($Q1$) and one with qualifiers ($Q4$).

In Figure 16(a), we show the evaluation times of query $Q1$ at each iteration of the experiment for algorithm PaX3. The top line in the graph shows the evaluation times of $Q1$ in the absence of XPath annotations, while the bottom line uses XPath annotations. Note that regardless of the presence of XPath annotations, tree fragmentation is beneficial since as fragmentation increases query evaluation time decreases, due to parallelism. The results also validate the analysis of Section 4 since the evaluation (parallel computation) cost of PaX3 depends only on the maximum fragment size. Since
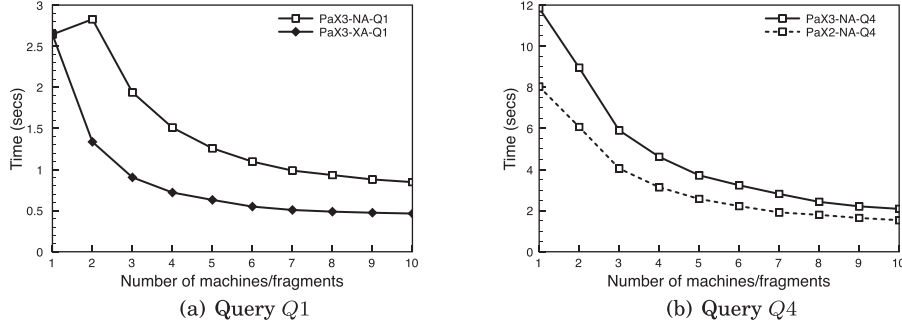
Fig. 16. Evaluation vs. Fragmentation.

the difference in maximum fragment sizes between iterations j and $(j+1)$ is $\frac{100}{j\times(j+1)}$ MB, the improvement in evaluation times between iterations starts to diminish after approximately iteration 6.

Let us now focus on the number of passes. Since query $Q1$ has no qualifiers, algorithm PaX3 can skip the first stage (pass) and only requires two passes of each fragment. Note that during the first iteration, when only one fragment exists (i.e., no actual fragmentation), PaX3 needs only to execute Stage 2 (a single pass) and can also skip the last stage, since any candidate answers from the second phase can be directly returned. When a second fragment is introduced in the next iteration of our experiment, PaX3 needs two passes per fragment. This additional pass causes a minor increase in evaluation time, as shown in the figure. The effect of parallelism outweighs however the cost of the additional pass from the third iteration on.

We now focus on algorithm PaX2. For query $Q1$, PaX2 has approximately the same evaluation time as algorithm PaX3, and thus it is not shown in the figure. To see why this is so, note that due to the lack of qualifiers in $Q1$, both algorithms require two passes over each fragment. The situation is different however, for a query like $Q4$, as shown in Figure 16(b). Due to qualifiers, algorithm PaX3 requires three passes per fragment, while PaX2 requires only two. The figure shows the savings coming from combining the first two passes of PaX3 into one pass in PaX2.

We now consider the effect of XPath-annotation optimization. XPath annotations are used in PaX3 to determine already at the second stage whether a candidate answer is a real answer. In light of this, as shown in Figure 16(a), the evaluation time of PaX3 is almost reduced by half. Thus we save the cost of Stage 3, which is now skipped not just in the first iteration but also in all subsequent ones. We also observe that XPath annotations do not alter the evaluation times of $Q4$ in each of PaX3 and PaX2. This is due to the '//' in the selection part of $Q4$, which, given the fragmentation in FT1, requires us to consider all the fragments.

Scalability with Data Size. This set of experiments is to study the scalability of the different algorithms (in terms of query evaluation times) with the size of data. Here we consider a more *natural* fragment tree for our data, shown to the right in Figure 15. The tree contains four XMark *"sites"* that are fragmented in different ways. Fragments F_0 (which includes the root of the whole tree) and F_3 contain two whole XMark *"sites,"* while the other two XMark *"sites"* are in fragments F_1 and F_2, and are further fragmented as shown in the figure. Unlike the previous experiment, not all fragments have the same size. The following table shows the approximate sizes of the various fragments in the first experiment iteration. Each fragment is assigned to a different machine and the cumulative size of the data is 100MB.
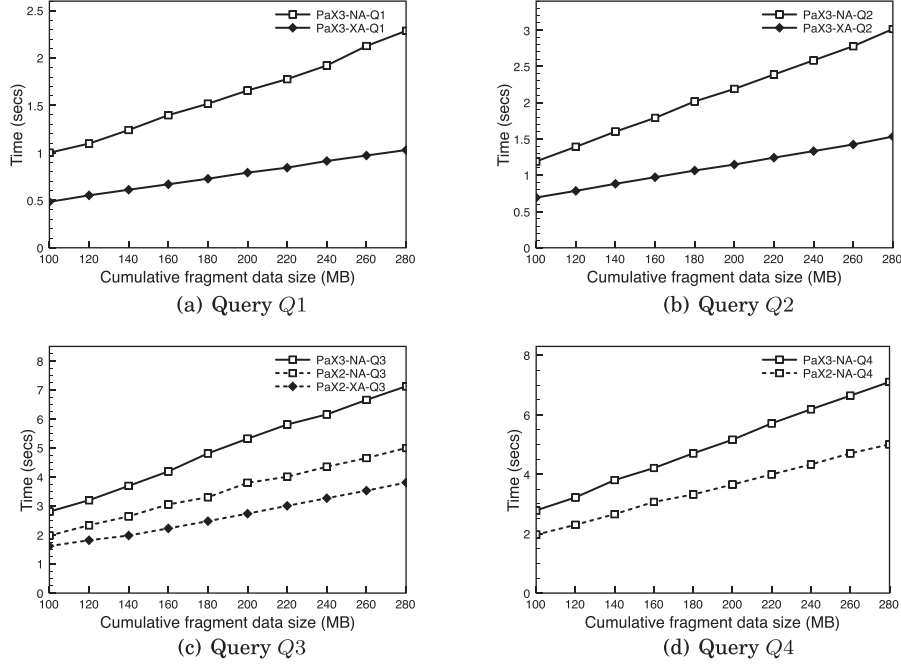
Fig. 17.   Evaluation time vs. Data scalability, for different queries.

Fragments	F_0, F_1, F_2, F_3	F_4, F_5, F_6, F_8	F_7	F_9
Size	5MB	12MB	28MB	8MB

At each iteration, we increase the size of each fragment, while maintaining constant the relative ratio of sizes between fragments. The increase is such that the cumulative size of data is augmented by 20MB, per iteration. At the last iteration, the tree is approximately 280MB. We consider four queries in this experiment, such that (a) two do not have qualifiers ($Q1$ and $Q2$), while the other two do ($Q3$ and $Q4$); (b) two are without a '//' in the selection part of the query ($Q1$ and $Q3$) while the other two do have a '//' ($Q2$ and $Q4$). Therefore, the queries cover all four possible combinations and are representative of a large class of common queries.

Figure 17(a) clearly shows that algorithm PaX3 scales linearly with the data size for query $Q1$ (with or without XPath annotations). The running times for PaX2 are almost identical with the two lines from PaX3, and therefore are not shown in the figure. As explained earlier, this is because both algorithms execute two passes over each fragment, due to the lack of qualifiers. The figure also illustrates that evaluation times are reduced by more than half when XPath annotations are used. More specifically, with the annotations in FT2, the evaluation only considers the data in fragments F_0, F_1, F_2 and F_3. Figure 17(b) shows similar results, even in the presence of '//'. Here in spite of '//' in the query, due to the fragmentation in FT2, only fragments F_0, F_1, F_2, F_3, F_6 and F_8 are considered during the evaluation.

Let us look now at Figure 17(c). Again, PaX3 scales linearly and moreover, its evaluation times are almost identical regardless of whether or not XPath annotations are in place. The reason for this is that here PaX3 must execute Phase 1 over *all* fragments, since $Q3$ has qualifiers. Our experiments show that the cost of evaluating qualifiers (Phase 1) is *dominant* in PaX3 and hence, gains from XPath annotations are minor for

(a) Query $Q1$

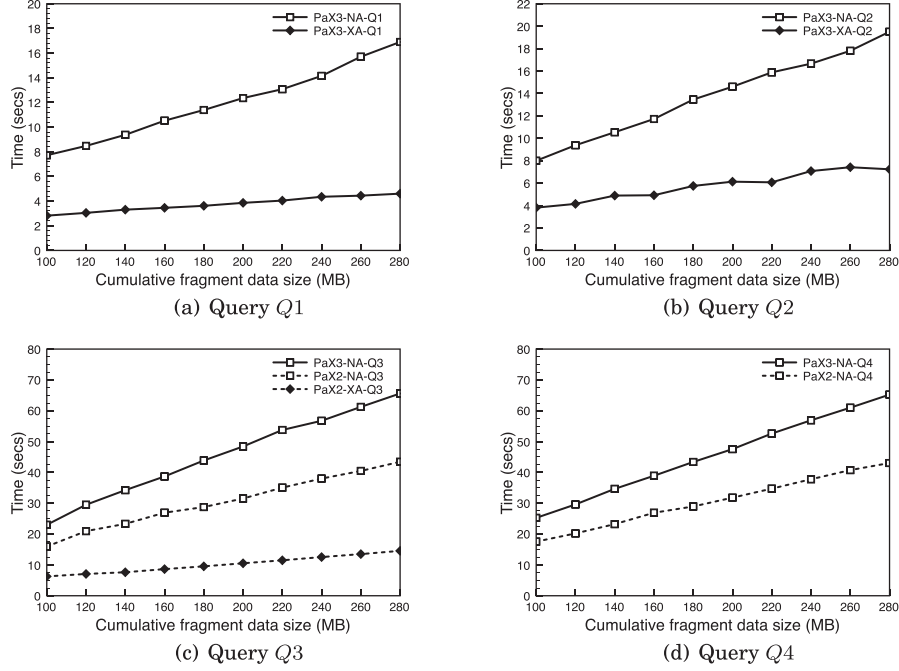(b) Query $Q2$

(c) Query $Q3$

(d) Query $Q4$

Fig. 18.   Total computation time for different queries.

this query, compared to the total execution time. Observe that this is not the case for queries like $Q1$ and $Q2$ where no qualifiers are present. There, the gains are significant, compared to the total execution time.

The second line in Figure 17(c) shows that PaX2 scales also linearly and is faster than PaX3, illustrating again the benefits of combining the two passes into one. Note that by using XPath annotations (third line in the figure), the evaluation time of PaX2 is improved even further. In contrast to PaX3, which computes qualifiers in all the fragments, PaX2 is more sophisticated in that it uses XPath annotations to decide on which fragments it executes the combined pass.

The last query considered, query $Q4$, has a '//' in its selection path and given the fragmentation of FT2, we must evaluate the query (and its qualifiers) over all the fragments of FT2. Here, XPath annotations do not help in ruling out any fragments. Therefore, as Figure 17(d) shows, the only gains in evaluation time are from combining the two passes of PaX3 into one pass of PaX2.

Total Computation Cost. This set of experiments are to show that our optimization method reduces in practice not only the parallel computation cost of our algorithms, but equally importantly, it also reduces the total computation cost. This experiment was built on the results of Experiment 2 and used exactly the same setting. To compute the total computation cost of the partial evaluation algorithms, we sum the evaluation times for each machine holding a fragment. Figure 18 shows the total computation cost for each query and algorithm of Figure 17.

Consider Figures 17(a) and 18(a). At first glance, the two figures seem similar, yet there is an important difference. By considering XPath annotations in the fragment tree while evaluating query $Q1$, the parallel computation cost was almost halved. Figure 18(a) shows that in addition, the total computation cost was reduced by two-thirds. This is because the machines holding those fragments identified irrelevant to

the query did not perform any computation. Therefore, by using XPath annotations not only the query was evaluated faster, but also it required less processing power to do so. Similarly, for $Q2$ XPath annotations saved two-thirds, in terms of parallel computation, and almost three-quarters, in terms of total computation.

Figures 18(c) and 18(d) illustrate that, in the absence of XPath annotations, in both PaX3 and PaX2 the savings in parallel computation are proportional to the corresponding savings in total computation cost. This is because without XPath annotations, both algorithms are evaluated over each fragment of the tree. However, in the presence of XPath annotations (last line in Figure 17(c) and 18(c)), the savings in total computation are even more significant than those in parallel computation.

Summary. We have shown that distributing tree fragments over various sites proves to be an effective strategy with significant reductions in evaluation time. Obviously, always using the (optimized) algorithm PaX2 along with XPath annotations is *sufficient* to consistently give the best results in terms of query evaluation time. We should temper the claim about the effectiveness of XPath annotations with the following observation. In the presence of '//' in the selection path of a query, annotations might not help much, as shown earlier for queries like $Q4$. However, it is not the case that the presence of '//' makes XPath annotations useless. As shown earlier for query $Q2$, if '//' appears after a prefix of the selection path of the query that matches a path in the XPath annotations, then a considerable number of fragments might be ruled out, thus improving query evaluation performance.

7.2. Experiments for the MapReduce Algorithm

Experimental Settings. We next evaluate the performance of Algorithm MRParBoX. We generated XML files of different sizes ranging from 10GB to 50GB using the standard XMark data generator. We used the partition method EKM [Kanne and Moerkotte 2006b] to partition each XML file into a number of fragments, each bounded by a threshold K (32MB by default). The queries used in this experiment are Boolean queries [Q] for queries Q given in Figure 14. For example, for $Q1$, [$Q1$] is [/*sites*/*site*/*people*/*person*]. Reported runtimes are averaged over multiple runs.

The experiments were conducted on Hadoop (see further on). All experiments are ran in a cluster with 64 computing nodes, each with a 2.63GHz CPU and 2GB RAM. The operating system is 64-bit Red Hat Enterprise Linux AS release 4.

EKM Partition Method. The EKM [Kanne and Moerkotte 2006b] method extends the KM method proposed in [Kundu and Misra 1977]. Given an XML tree T and a threshold K, EKM works by processing the nodes of T bottom-up. When processing a node v whose subtree is larger than K, EKM chooses the largest child of v to construct a new fragment for the subtree.

Hadoop. Hadoop[1] is an open-source implementation of the MapReduce framework in Java, provided by Google. We used Hadoop 0.21.0, implemented MRParBoX in C++, and ran MRParBoX by using Hadoop Streaming,[2] which is a utility that supports creating and running Map/Reduce jobs written in C++ as the mapper or the reducer.

To present the experimental results, we briefly review the flow of Hadoop streaming. Suppose that XML fragments have been organized as <*key*, *value*> pairs and uploaded to the DFS. Given the fragments and some query as inputs, Hadoop system automatically executes MRParBoX as follows. (1) For each XML fragment file in
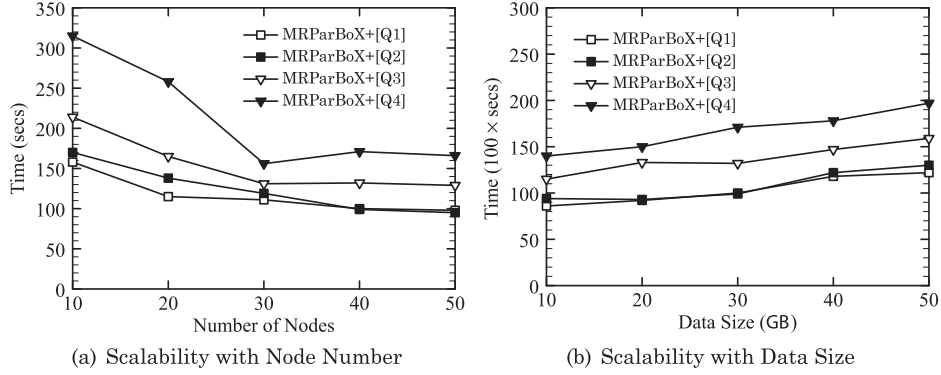
---

[1]http://hadoop.apache.org
[2]http://wiki.apache.org/hadoop/HadoopStreaming

Fig. 19. Scalability of MRParBoX with Node Number and Data Size.

$<key, value>$, Hadoop distributes it to some map node based on the hash value of *key*, and then the corresponding node invokes the map function and computes the output. (2) Then, Hadoop sends the outputs of the map nodes to the reduce nodes. Before invoking the reduce function, at each reduce node, Hadoop sorts the map outputs received. This step is called *shuffle*. (3) Finally, taking the sorted map outputs as input, a reduce node generates the final result. We use T_{map} to represent the time cost of the map step, $T_{shuffle}$ for the *shuffle* step, T_{reduce} for the reduce step, and T_{total} for the runtime of the entire MRParBoX process. MRParBoX needs only one reduce node, and reduce runs after inputs have been received from all the map nodes.

Scalability with the Number of Computing Nodes. This set of experiments studies the scalability of MRParBoX with the number of computing nodes. Fixing data size to 30GB, we varied the number of computing nodes from 10 to 50. We stored the data in the DFS of Hadoop, and invoked MRParBoX to evaluate the Boolean forms of the queries given in Figure 14. The results are shown in Figure 19(a). The results tell us that when the number of computing nodes was increased, it took less time to evaluate the queries. This is because (a) the computing tasks of map were divided into several parts and evaluated in parallel, resulting in significant saving in the T_{map} cost; and (b) as the node number increased, the data transferring operations in the shuffle procedure distributed inputs by starting multiple HTTP threads that were executed in parallel; this reduced the $T_{shuffle}$ cost. Note that when the number of computing nodes is beyond 30, adding more nodes does not yield significant improvement in evaluation time. This is because MRParBoX uses a single reducer, which communicates with all map nodes and hence, compromises the gains when the number of nodes increases. It should be remarked that T_{reduce} remained the same when the number of nodes varied since we fixed the data set in this experiment.

Scalability with Data Size. This set of experiment is to study the scalability of MRParBoX with the size of data. We configured Hadoop using 40 nodes, varied the data size from 10GB to 50GB, where each dataset was partitioned into 32MB fragments. We used the same set of Boolean queries as before. As shown in Figure 19(b), the runtime increased linearly with the data size. The reason is as follows. For a query, (1) the cost of map, T_{map}, mainly depends on the number of fragments processed for the query, which is linear in the size of the data, (2) the cost of *shuffle* between computing nodes, $T_{shuffle}$, mainly depends on the number of query vectors produced by map function, which is also linear with the data size, and (3) the cost of reduce is linear with the number of the fragments.

(a) Scalability with Node Number and Data Size   (b) Effect of Fragment Size on MRParBoX
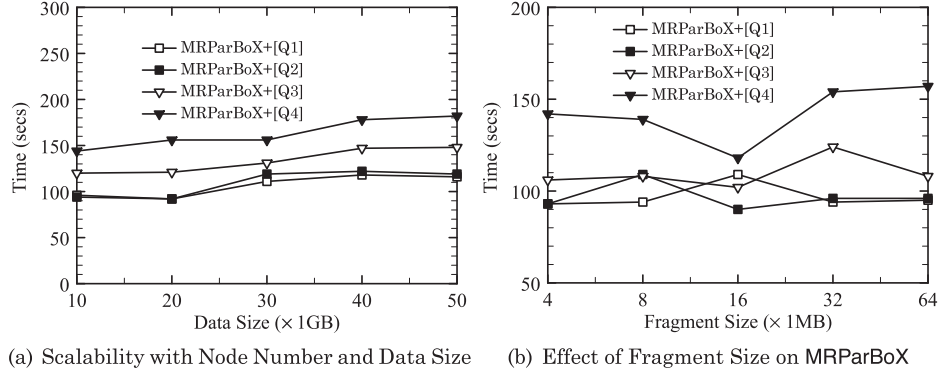
Fig. 20. Scalability of MRParBoX and Effects of Fragment Size on MRParBoX.

Scale Up of MRParBoX. This set of experiments is to study the scalability of MR-ParBoX with both the data size and the number of computing nodes. Fixing the ratio between data size (GB) and the number of nodes as 1, we varied the data size from 10GB to 50GB, and thus the number of nodes from 10 to 50. The runtime of MRParBoX is shown in Figure 20(a). When data size and node number increase in the same ratio, the time cost of MRParBoX increases very slightly. The slight increase is because (a) a single reduce node is used for the reduce step and its runtime increases with the size of the data set, although map and *shuffle* procedures can be executed in parallel; and (b) the cost of managing the computing nodes increases with the number of nodes.

Effects of Fragment Size. This set of experiments is to study the effect of the threshold K for *fragment size* on the performance of MRParBoX. We fixed the data size at 10GB, and used EKM to partition the data into fragments of different sizes by varying K from 4MB to 64MB. Figure 20(b) shows the results on the four queries when we varied the threshold K. Observe that the fragment sizes affect the performance of MRParBoX, and there exists no optimal fragment size for all queries. This is because the fragmentation and query together determine which fragments need to be visited for the query. For instance, when K is 16MB, Q_2–Q_4 performed the best but Q_1 showed the worst performance. This is because Q_1 was not very selective on some map node to which several fragments are mapped, each of which is less than 16MB but the sum of their sizes is about 16MB.

Summary. The results of this set of experiments have verified that as a natural extension of ParBoX, MRParBoX is able to process queries on very large XML data, and scales well with the size of data set. In other words, partial evaluation can be readily incorporated into the MapReduce framework.

7.3. Experiments for Centralized Evaluation Algorithms

Experimental Settings. We next evaluate Algorithms BT-CPax and TT-CPax, as well as two of their optimized versions that use the annotation-based optimization, denoted by BT-CPax-XA and TT-CPax-XA, respectively.

We generated several XML files of different sizes ranging from 500MB to 4GB using the standard XMark data generator. The partition method EKM [Kanne and Moerkotte 2006b] was used to partition each XML file into a number of fragments whose sizes were bounded by a threshold K (256KB by default). In this set of experiments we used the same queries given in Figure 14. Reported runtimes are averaged over multiple runs of experiments. We implemented BT-CPax and TT-CPax with egcs2.91.57 C++,
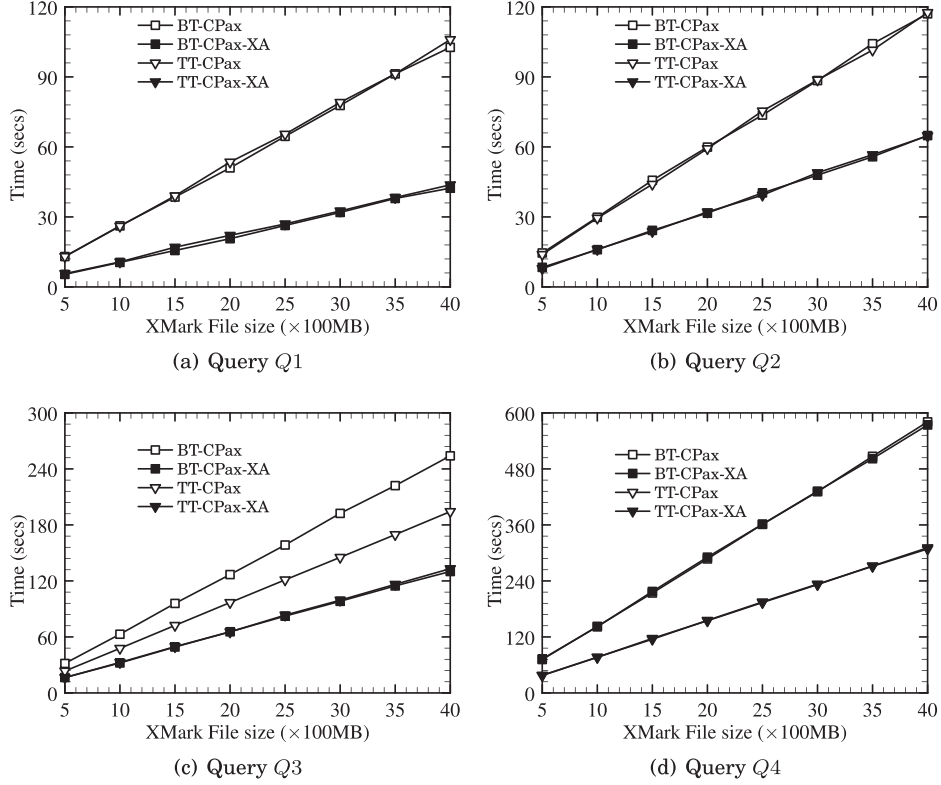
Fig. 21. Performance of the four centralized algorithms on different queries.

and conducted the experiments on a 64-bit Windows machine with 2.93GHz Pentium Dual CPU and 2GB memory.

Efficiency of BT-CPax *and* TT-CPax. In this set of experiments, we varied the data size from 500MB to 4GB. Figure 21 shows the runtime of the four different algorithms on the four queries of Figure 14. These results clearly show the benefit of TT-CPax over BT-CPax. We find that for queries with qualifiers (i.e., $Q3$, $Q4$), TT-CPax outperforms BT-CPax, and TT-CPax-XA outperforms BT-CPax-XA (see Figures 21(c) and 21(d)). That is because BT-CPax needs to evaluate the qualifiers of a query at each node of an XML tree, while TT-CPax evaluates selection path and qualifiers in a single pass, and uses the results of selection path evaluations to guide whether to evaluate subqualifiers at a node. This also explains why TT-CPax-XA outperforms BT-CPax-XA. We also find that for queries without qualifiers i.e., $Q1$, $Q2$), BT-CPax and TT-CPax perform the same, and so do BT-CPax-XA and TT-CPax-XA (see Figures 21(a) and 21(b)). In fact, BT-CPax and TT-CPax (resp. BT-CPax-XA and TT-CPax-XA) are essentially the same when queries do not have qualifiers.

The experimental results also tell us that TT-CPax-XA (resp. BT-CPax-XA) significantly improve the performances of TT-CPax (resp. BT-CPax), showing the utility of the annotation-based optimization.

Scalability with Data Size. The proposed centralized algorithms aim to process very large XML documents. Figure 21 shows that when we vary the size of data

(a) Query $Q1$

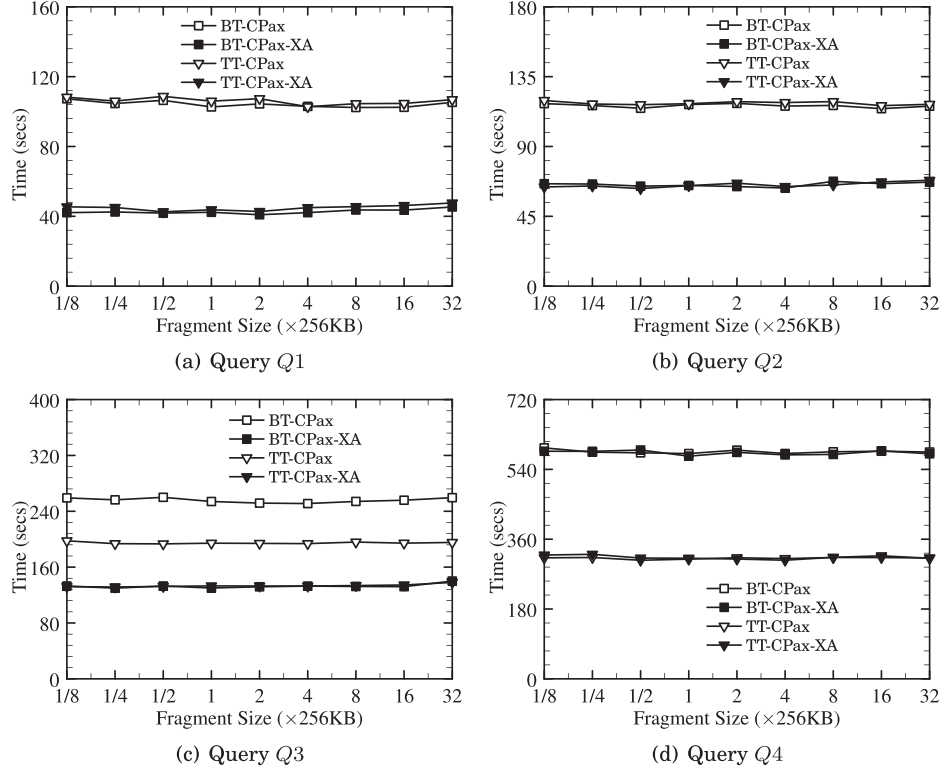(b) Query $Q2$

(c) Query $Q3$

(d) Query $Q4$

Fig. 22. Performance of centralized algorithms when varying fragment size.

from 500MB to 4GB, all the algorithms scale linearly with the size of data on the four queries, including the complicated query $Q4$.

Effects of Fragment Size. This set of experiments is to study the effect of the threshold K for fragment size on the performance of the four algorithms. Fixing the data size at 4GB, we varied K from 32KB to 8MB. Figure 22 shows the results on the four queries when we varied the threshold K. Observe that the performance of the algorithms is not very sensitive to the fragment size, and all the four algorithms achieve the best performance on all the four queries when the *fragment size* is in the range between 32KB and 8MB. When the fragment size is small, both the selection path and annotation-based optimization can be more effective in pruning (at least part of) a fragment, thus reducing the I/O cost. However, smaller fragments also increase the cost of managing the fragment tree and loading fragments. On the other hand, if the fragment size is large, the chance of disregarding an entire fragment is smaller, and thus it would be more likely to process the entire XML tree, while the cost of managing the fragment tree is lower.

Comparison with Other Query Engines. This set of experiments aims to compares our centralized algorithms with three well-known XML query engines, namely, Saxon [Saxon], Galax [Galax] and MonetDB/XQuery [MonetDB]. Among the three XML query engines, Saxon and Galax focus mainly on the evaluation of queries over *file-based* XML systems as does our approach, where XML documents are stored natively in a file system, and queries are evaluated over the individual XML files. In contrast,

$Q5$	`/site/catgraph/edge`
$Q6$	`/site/regions/africa//name`
$Q7$	`/site/catgraph/edge[from='category7' or(not (to='category5'))]`

Fig. 23. Additional Sample Queries.

MonetDB/XQuery uses a more database-centric approach where XML documents are *imported* into the MonetDB system by being shredded into relational databases used in MonetDB. As remarked in Section 1, file-based XML systems are particularly useful in domains like life sciences (e.g., biology), astronomy, and even in commonly used tools like Microsoft Office, where users would prefer to use simple file-based systems rather than complicated full-fledged databases. Hence, next we first compare with the two file-based systems, and then with the database-centric MonetDB/XQuery system.

Comparison with File-Based Query Engines. In this experiment, we compare the proposed centralized algorithm with Saxon-HE 9.2 and Galax 1.0.1. Again, we used XMark data of different sizes ranging from 100MB to 500MB. In addition to the queries of Figure 14, we considered three new queries $Q5$–$Q7$ given in Figure 23, which are more selective than those in Figure 14. For all the three systems, we measure the *end-to-end* evaluation time. For Saxon and Galax, this includes the time of loading an XML file into memory and the query execution time. For our own algorithms, the *end-to-end* evaluation time includes the time of partitioning the XML file (using the default threshold 256KB as the fragment size), the time of loading fragments into memory, and the query evaluation time. We only report the time cost on TT-CPax-XA since it outperforms the other three proposed centralized algorithms. Note that we can eliminate the partitioning time from the *end-to-end* if the input XML file has already been partitioned, and the *end-to-end* in this case is denoted by "*loading+query*."

The results of comparing TT-CPax-XA with Saxon are shown in Table I. When the dataset is larger than 300M, Saxon fails to load XML files into tree because it runs out of memory. On XML files of sizes 100-300M, Saxon outperforms TT-CPax-XA in terms of the *end-to-end* time. However, note that the *end-to-end* time of TT-CPax-XA includes the partitioning time (including reading an XML file, partitioning it and writing the generated fragments to the disk), which dominates the runtime of TT-CPax-XA. If we take out the partitioning time, the running time of TT-CPax-XA is the "*loading+query*," which is actually the counterpart of the *end-to-end* time of Saxon. We can also see that in terms of the "*loading+query*," TT-CPax-XA runs faster on $Q5$–$Q7$, which are more selective than $Q1$–$Q4$.

Table II shows the result of Galax on 100MB data. It fails to run on 200M data due to running out of memory. We can see that its performance is worse than Saxon and TT-CPax-XA by comparing the results in Table II and Table I.

Comparison with XML Database. We next compare the centralized algorithm with MonetDB/XQuery 4.0 64-bit version compiled with 64 bit oid. Again we used XMark data of different sizes ranging from 500MB to 4GB, and the seven queries given in Figures 14 and 23.

The experimental results are shown in Table III. We report both *end-to-end* time and *query/loading+query* time for MonetDB and TT-CPax-XA. The *end-to-end* time of MonetDB includes the query time and the time of importing XML file into MonetDB. The *end-to-end* time of TT-CPax-XA is the same as we use to compare with other *file-based* systems. The *query* time for MonetDB is the time of answering a query, excluding the time of importing XML file. The *loading+query* time for TT-CPax-XA includes the time for loading fragments into memory and the query evaluation time.

Table I. Comparison with Saxon

Query			Runtime (Second)				
			100MB	200MB	300MB	400MB	500MB
$Q1$	TT-CPax-XA	*loading+query*	1.09	2.14	3.29	4.55	5.7
		end-to-end	14.09	27.12	44.29	59.55	82.7
	Saxon	*query execution*	0.31	0.56	1.2	–	–
		end-to-end	4.42	8.88	14.8	–	–
$Q2$	TT-CPax-XA	*loading+query*	1.54	3.22	5.15	6.53	7.85
		end-to-end	14.54	28.22	46.15	61.53	84.85
	Saxon	*query*	0.34	0.67	1.44	–	–
		end-to-end	4.47	8.88	14.92	–	–
$Q3$	TT-CPax-XA	*loading+query*	3.26	6.56	9.7	13.02	16.39
		end-to-end	16.26	31.57	50.7	68.02	93.39
	Saxon	*query*	0.14	0.2	0.44	–	–
		end-to-end	4.27	8.3	13.95	–	–
$Q4$	TT-CPax-XA	*loading+query*	7.41	14.58	22.58	30.75	37.71
		end-to-end	20.41	39.58	63.58	85.75	114.71
	Saxon	*query*	0.19	0.31	0.59	–	–
		end-to-end	4.33	8.41	14.11	–	–
$Q5$	TT-CPax-XA	*loading+query*	0.08	0.12	0.31	0.34	0.36
		end-to-end	13.08	25.13	41.31	55.34	77.36
	Saxon	*query*	0.03	0.05	0.22	–	–
		end-to-end	4.16	8.34	13.84	–	–
$Q6$	TT-CPax-XA	*loading+query*	0.17	0.34	.41	.59	.42
		end-to-end	13.17	25.36	41.41	55.59	77.42
	Saxon	*query*	0.03	0.03	0.17	–	–
		end-to-end	4.16	8.11	13.64	–	–
$Q7$	TT-CPax-XA	*loading+query*	0.13	0.16	0.31	0.42	0.42
		end-to-end	13.12	25.15	41.31	55.42	77.42
	Saxon	*query*	.05	.05	.09	–	–
		end-to-end	4.57	8.16	13.44	–	–

Table II. Runtime of Galax

end-to-end	Q1	Q2	Q3	Q4	Q5	Q6	Q7
Time on 100MB (secs)	198	521	121	17,400	123	115	115

In terms of the *end-to-end* time, TT-CPax-XA is almost always better than Mon-etDB/XQuery on all queries for datasets of 1GB or larger. Apart from the time reported in Table III, we also evaluated the *end-to-end* time on larger data set. For example, on 10GB XMark data, we find that the *end-to-end* time of $Q1$ for MonetDB/XQuery is 13,291 *seconds*, which is about 80 times of the time for 1GB data and 9 times of the time for 4GB data. In contrast, TT-CPax-XA needs 1,652 *seconds*, and it increases linearly with the size of XML files. We note that the importing time (shredding XML files into MonetDB) dominates the *end-to-end* time of MonetDB while the partitioning step dominates TT-CPax-XA.

Purely in terms of evaluation time, MonetDB performs better than TT-CPax-XA for $Q1$–$Q4$. On queries $Q5$–$Q7$, TT-CPax-XA performs similar or better than does

Table III. Comparison with MonetDB/XQuery

Query			Runtime (Second)							
			0.5GB	1GB	1.5GB	2GB	2.5GB	3GB	3.5GB	4GB
$Q1$	TT-CPax-XA	*loading+query*	5.7	10.71	17.03	22.11	27	32.54	38.32	43.75
		end-to-end	82.7	164	248	330	412	494	577	659
	MonetDB/	*query*	1.50	2.41	3.34	4.08	9.63	6.06	6.54	7.35
	XQuery	*end-to-end*	61.4	166	313	431	852	1054	1233	1476
$Q2$	TT-CPax-XA	*loading+query*	7.85	16.08	23.77	32.13	39.32	48.95	56.66	64.73
		end-to-end	84.8	170	255	340	424	511	596	680
	MonetDB/	*query*	1.23	1.64	2.32	2.56	8.12	3.43	3.9	4.32
	XQuery	*end-to-end*	61.2	165	312	430	851	1052	1231	1474
$Q3$	TT-CPax-XA	*loading+query*	16.39	32.64	49.59	65.38	83.3	99.4	116	133
		end-to-end	93.4	187	281	373	468	561	655	749
	MonetDB/	*query*	5.74	9.06	13.45	16.8	29	24.87	27.83	32.86
	XQuery	*end-to-end*	65.7	172	323	444	871	1073	1255	1502
$Q4$	TT-CPax-XA	*loading+query*	37.71	76.29	117	155	194	232	272	311
		end-to-end	115	230	348	463	579	694	811	927
	MonetDB/	*query*	5.73	9.07	13.54	16.92	35.96	25.61	27.86	32.23
	XQuery	*end-to-end*	65.65	172	323	444	878	1074	1255	1501
$Q5$	TT-CPax-XA	*loading+query*	0.36	0.79	1.22	1.59	2.07	2.44	3.07	3.07
		end-to-end	77.36	155	232	310	387	464	542	619
	MonetDB/	*query*	1.29	1.66	2	2.36	8.84	3.34	3.67	4.17
	XQuery	*end-to-end*	61.21	165	311	429	851	1052	1231	1473
$Q6$	TT-CPax-XA	*loading+query*	0.42	1.36	2.3	2.61	3.54	4.31	4.98	5.73
		end-to-end	77.42	155	233	311	389	466	544	622
	MonetDB/	*query*	1.13	1.49	2.16	2.4	10.03	3.34	3.73	4.26
	XQuery	*end-to-end*	61.05	165	311	429	852	1052	1231	1473
$Q7$	TT-CPax-XA	*loading+query*	0.42	1.02	1.49	1.9	2.34	2.96	3.44	3.73
		end-to-end	77.42	155	232	310	387	465	542	620
	MonetDB/	*query*	1.42	2.14	2.65	2.88	12.37	3.91	4.7	5.02
	XQuery	*end-to-end*	61.34	166	312	430	855	1052	1232	1474

MonetDB/XQuery. This is because the selectivity of those queries are smaller than those of the first four queries $Q1$–$Q4$, and thus TT-CPax-XA needs to scan fewer nodes and fragments in precessing the queries. The selectivity here can be understood as the number of nodes that need to be visited to answer a query, which mainly depends on two factors. For queries having no "//" operators or qualifiers, it mainly depends on the result size, while for the queries containing qualifiers or "//" operators, the sizes of the subtree involved can reflect selectivity better. Consider the same query setting for 100MB data (the case of 100MB can explain larger data as well, since they are all generated by the standard Xmark generator). (1) For $Q1$ and $Q5$, the result sizes can reflect the selectivity well since there are no qualifiers or "//." Thus we count the result sizes of $Q1$ and $Q5$ on XMark100M data: the result sizes of $Q1$ and $Q5$ are 25,500 and 1,000, respectively. The result size of $Q5$ is much smaller than that of $Q1$, that is, TT-CPax-XA needs to visit fewer nodes to answer $Q5$ than $Q1$. This is why TT-CPax-XA performs better on $Q5$ than $Q1$. (2) For $Q2$ and $Q6$, the subtree sizes of *open_auctions* and *africa* can reflect the selectivity well, respectively. We count their subtree sizes and get (*open_auctions*, 675,814) and (*africa*, 17,451). For these two queries, TT-CPax-XA would need to traverse the whole subtrees under these two elements respectively, and the subtree size of *africa* is much smaller than the one of *open_auctions*. This explains why TT-CPax-XA performs better on $Q6$ than $Q2$.

Summary. From the experimental results we find the following. (1) Our centralized algorithms BT-CPax and TT-CPax are able to process queries on large XML data, and scale well the size of the dataset. (2) TT-CPax usually outperforms BT-CPax, and XPath annotations are effective in improving the performance of both BT-CPax and TT-CPax. (3) TT-CPax performs better than file-based query engines Saxon-HE 9.2 and Galax 1.0.1 on large datasets. While Saxon-HE 9.2 and Galax 1.0.1 may fail since they are not able to load data into memory, TT-CPax is not hampered by the limitation of memory capacity. (4) Compared with MonetDB/XQuery 4.0 64-bit version, while MonetDB does better in query evaluation time, TT-CPax takes less *end-to-end* time, and has shown promises as an effective method for querying XML files, which has a wide range of applications.

8. RELATED WORK

This work is related to previous research in the areas of partial evaluation, XPath query evaluation in the distributed setting, and XPath queries on large XML data.

Partial evaluation. Partial evaluation has found a wide range of applications from compiler optimization to parallel evaluation of functional programming languages [Jones 1996]. Its relevance to query evaluation has surfaced from time to time, most notably in the Disco system [Tomasic et al. 1996] and in query rewriting with views and deductive databases [Godfrey and Gryz 2000; Gupta et al. 1994]. From a traditional functional programming perspective, our work is most closely related to the uses of partial evaluation in dataflow architectures [Jones 1996] in which one evaluates some or all of the arguments of a function in parallel. The difference from our work is that traditional functional programming neglects the benefits of partial evaluation for functions accessing large data sets.

To our knowledge, partial evaluation was first explored to process (a) Boolean XPath queries [Buneman et al. 2006] and (b) data-selecting XPath queries [Cong et al. 2007], in the distributed setting. These have led to the algorithms of Sections 3 and 4, respectively, which are among the first algorithms with performance guarantees on the number of visits, the network traffic and the computation costs.

Distributed XPath query processing. Close to this work are [Amer-Yahia et al. 2004; Kanne et al. 2005; Suciu 2002]: [Suciu 2002] for distributed (data-selecting) query evaluation on semistructured data, [Kanne et al. 2005] for evaluating XPath queries without qualifiers on tree fragments distributed over disk pages, and [Amer-Yahia et al. 2004] for processing (aggregate and Boolean) queries on hierarchical distributed catalogs [Smith and Howes 1997]. Some of these algorithms also have nice performance bounds on total network traffic, computation and communication steps. These algorithms differ from our work mostly in technical approaches. More specifically, [Amer-Yahia et al. 2004; Kanne et al. 2005; Suciu 2002] employ query decomposition and focus on query plan generation, which rewrite an input query into subqueries appropriate for individual sites (using, e.g., the accessibility of the distributed data). In contrast, we avoid this overhead by sending the whole query to each relevant site. In addition, our algorithms characterize partial answers as Boolean expressions rather than concrete data as found in the previous work. Recently, a technique for generating distributed execution plans for a vertically partitioned and distributed XML document is proposed [Kling et al. 2010], which considers XPath queries similar to queries studied in this work. An XPath query is decomposed into a number of local subqueries, each for an involving site. Given a subquery, each site generates a local execution plan and estimate the cost, and then the central node constructs a distributed execution plan

that determines how the local results are combined to produce the final query answer. However, when dependency exists among the query results of different subqueries, the subqueries may have to be evaluated sequentially.

Another active research area on distributed XML query processing has been query language extensions that include communication and distribution primitives for querying heterogeneous XML data sources. XRPC [Zhang and Boncz 2007] enhances the existing concept of XQuery functions with the Remote Procedure Call (RPC) paradigm, thus querying heterogeneous XML data sources in multiple sites. Active XML (AXML) [Abiteboul et al. 2008] allows us to embed in XML documents with calls to Web or AXML service functions. The evaluation of a service call results in an XML fragment, which is inserted into the original XML document, and gets reevaluated. A lazy evaluation algorithm [Abiteboul et al. 2004] is proposed to efficiently evaluate queries over AXML documents. Instead of materializing all the service calls embedded in an AXML document, the proposed lazy algorithm identifies a tight superset of service calls that should be materialized to answer a query. As observed by [Kling et al. 2010], these approaches cater primarily to a data integration scenario. AXML, for instance, distinguishes static data, that is, data that are relatively stable, from dynamic data, that is, data that are updated frequently. There language extensions usually need to explicitly specify the locations for query operators that need to be evaluated in remote sites. In contrast, we process standard XPath queries on data that are arbitrarily partitioned and distributed, a setting different from the settings of those extensions.

In addition to explicitly using URLs to reference data sources in an XML query, Mutant query plans (MQPs) [Papadimos and Maier 2002] can refer to abstract resource names (URNs). Mutant query plans are transferred as XML documents. Every site maintains a local catalog that maps each URN to either a URL, or to a set of sites that know more about the URN.

These distributed XPath query processing approaches are not capable of evaluating distributed queries in parallel when the evaluation in a site depends on the results from other sites. In contrast, the proposed partial evaluation techniques enable the parallel evaluation of all subqueries over all fragments. It would be interesting to explore the possibility of incorporating the idea of partial evaluation into the existing proposals on distributed XML query processing.

There has been a large body of work on distributed query processing (see, e.g., Kossman [2000] for a nice survey). The focus has mostly been on techniques for minimizing communication cost, via hybrid shipping, two-phase optimization [Kossman 2000], replication [Abiteboul et al. 2003], and parallel query evaluation [DeWitt and Gray 1992; Hsiao and DeWitt 1990]. This work aims to minimize both data movement and communication steps by shipping residual functions (Boolean formula) rather than data. This idea of partial evaluation may also be combined with recent techniques developed for P2P systems [Crainiceanu et al. 2004; Ganesan et al. 2004; Halevy et al. 2004; Jagadish et al. 2005]. In particular, Bonifati and Cuzzocrea [2006] studied how to store and retrieve XML data over structured Peer-to-Peer (P2P) networks. A document is split into various fragments, which are locally stored at peers based on their path expressions, that is, the path reaching a fragment. The path expressions of fragments are also used in evaluating XPath queries as we use annotation to filter fragments. XPath annotation is also explored in Zhang et al. [2009] and Marian and Siméon [2003] for query processing.

Centralized XPath query processing. A number of algorithms have been developed for evaluating XPath queries in centralized systems. One line of work is to store and query XML data using RDBMS [Florescu and Kossmann 1999; MonetDB; Pal et al.

2004; Schmidt et al. 2000; Zhang et al. 2001]. The main idea is to store XML data as relations in RDBMS, and translate XML queries into SQL. These algorithms usually perform well, notably MonetDB [Boncz et al. 2006; Schmidt et al. 2000]. Another approach is to develop native XML stores, [Al-Khalifa et al. 2002; Bruno et al. 2003; Colen et al. 2002; Gottlob et al. 2002; Kanne et al. 2005; Kaushik et al. 2002; Koch 2003; Lu et al. 2005; Milo and Suciu 1999; Ramanan 2002; Zhang et al. 2004], which have produced XPath optimizations and indexing techniques that are complementary to this work. In view that XQuery is used to query not only XML data in databases, but also in applications that process XML files, Marian and Siméon [2003] identifies which parts of an input document are needed to answer an XQuery to overcome the memory limitation. Partial evaluation provides another effective approach to processing queries on XML files.

Also related to this work are partition-based methods [Huck et al. 1999; Kanne and Moerkotte 2006b] for storing XML data. While those methods aim to reduce the I/O cost, they often require to load some fragments multiple times when processing queries (due to the interdependency between fragments), and hence, incur extra I/O overhead. In contrast, based on partial evaluation, our centralized evaluation algorithms visit each fragment at most twice, irrespective of how an XML tree is fragmented and stored.

Our centralized query evaluation algorithms leverage prior work on partitioning XML data. Several partitioning methods are already in place. (a) In [Bordawekar and Shmueli 2004], XML partitioning was treated as a traditional (NP-complete) clustering problem, for which a pseudo-polynomial algorithm was developed [Lukes 1974]. (b) A partitioning algorithm, called KM, was proposed, which processes the nodes of an XML tree in a bottom-up fashion. Whenever the subtree under the current node is larger than a given threshold value, the algorithm creates a fragment for the largest subtree under the current node. A variation of KM, known as EKM, was proposed in Kanne and Moerkotte [2006b] that adopts depth-first traversal of XML trees instead. (c) An algorithm based on dynamic programming, referred to as GHDW, was given in Kanne and Moerkotte [2006b], which aims to find a partition with the fewest fragments. (d) Another approach [Kanne and Moerkotte 2006a], called RS, generates a subtree from right to left whenever the size of the subtree is larger than a predefined threshold. We adopted EKM in our experimental study.

9. CONCLUSIONS

We have proposed the first distributed evaluation algorithms for Boolean XPath queries and for data-selecting XPath queries based on partial evaluation, with *performance guarantees* on network traffic, total computation and communication steps. We have also shown that partial evaluation can be readily incorporated into the MapReduce framework, by presenting a MapReduce algorithm for processing Boolean XPath queries using partial evaluation. In addition, we have developed centralized algorithms for processing XPath queries on large XML documents, based on partial evaluation. We have shown analytically and experimentally that our techniques are scalable and efficient for handling complex XPath queries on large datasets. In light of these results, we contend that partial evaluation is effective in processing XML queries in both distributed systems and in centralized systems.

There is naturally much more to be done. First, we plan to extend partial evaluation to processing other XML queries, such as queries with other axes and extensions of twig queries with multiple returning nodes. It is known that XPath queries with axes for upward traversals (i.e., parent axis and ancestor axis) can be converted to equivalent XPath queries with downward traversals only (with child axis and descendant axis), when the queries are evaluated at the root of any XML tree [Benedikt et al. 2005]. As a result, the algorithms developed in this work can be readily applied to XPath

queries with upward traversals. Furthermore, the algorithms can be extended to support sibling axes in the same framework, with only extra engineering work. To process extensions of twig queries with multiple returning nodes, the proposed centralized algorithms can be easily extended without extra overhead by treating the twig branches as predicates. In the distributed setting, when the leaf nodes of twig patterns are found on distinct sites, we can retrieve them by one more visit of each involved site. Second, it is interesting to find out whether the technique can be used to support XML updates, in the distributed setting or when large XML data is distributed across different disk pages. Finally, it is possible to integrate partial evaluation with other optimization techniques for XML query processing.

REFERENCES

ABITEBOUL, S., BENJELLOUN, O., MANOLESCU, I., MILO, T., AND WEBER, R. 2002. Active XML: Peer-to-peer data and web services integration. In *Proceedings of the International Conference on Very Large Databases*. 1087–1090.

ABITEBOUL, S., BONIFATI, A., COBENA, G., MANOLESCU, I., AND MILO, T. 2003. Dynamic XML documents with distribution and replication. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 527–538.

ABITEBOUL, S., BENJELLOUN, O., CAUTIS, B., MANOLESCU, I., MILO, T., AND PREDA, N. 2004. Lazy query evaluation for active XML. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 227–238.

ABITEBOUL, S., BENJELLOUN, O., AND MILO, T. 2008. The active XML project: An overview. *VLDB J. 17*, 1019–1040.

AL-KHALIFA, S. A., JAGADISH, H. V., KOUDAS, N., PATEL, J. M., SRIVASTAVA, D., AND WU, Y. 2002. Structural joins: A primitive for efficient XML query pattern matching. In *Proceedings of the International Conference on Data Engineering*. 141.

ALTINEL, M. AND FRANKLIN, M. J. 2000. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of the International Conference on Very Large Databases*. 53–64.

AMER-YAHIA, S., SRIVASTAVA, D., AND SUCIU, D. 2004. Distributed evaluation of network directory queries. *IEEE Trans. Knowl. Data Engin. 16*, 4, 474–486.

BENEDIKT, M., FAN, W., AND KUPER, G. M. 2005. Structural properties of XPath fragments. *Theor. Comput. Sci. 336*, 1, 3–31.

BONCZ, P. A., GRUST, T., VAN KEULEN, M., MANEGOLD, S., RITTINGER, J., AND TEUBNER, J. 2006. MonetDB/XQuery: A fast XQuery processor powered by a relational engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 479–490.

BONIFATI, A. AND CUZZOCREA, A. 2006. Storing and retrieving XPath fragments in structured P2P networks. *Data Knowl. Eng. 59*, 247–269.

BORDAWEKAR, R. AND SHMUELI, O. 2004. Flexible workload-aware clustering of XML documents. In *Proceedings of the International XML Database Symposium*. 346–357.

BREMER, J. AND GERTZ, M. 2003. On distributing XML repositories. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*. 73–78.

BRUNO, N., GRAVANO, L., KOUDAS, N., AND SRIVASTAVA, D. 2003. Navigationvs index-based XML multi-query processing. In *Proceedings of the International Conference on Data Engineering*. 139–150.

BRUNO, N., KOUDAS, N., AND SRIVASTAVA, D. 2002. Holistic twig joins: Optimal XML pattern matching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 310–321.

BUNEMAN, P., CONG, G., FAN, W., AND KEMENTSIETSIDIS, A. 2006. Using partial evaluation in distributed query evaluation. In *Proceedings of the International Conference on Very Large Databases*. 211–222.

COLEN, E., KAPLAN, H., AND MILO, T. 2002. Labeling dynamic XML trees. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*. 271–281.

CONG, G., FAN, W., AND KEMENTSIETSIDIS, A. 2007. Distributed query evaluation with performance guarantees. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 509–520.

CRAINICEANU, A., LINGA, P., GEHRKE, J., AND SHANMUGASUNDARAM, J. 2004. Querying peer-to-peer networks using P-Trees. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*. 25–30.

DEAN, J. AND GHEMAWAT, S. 2004. MapReduce: Simplified data processing on large clusters. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*. 137–150.

DEWITT, D. J. AND GRAY, J. 1992. Parallel database systems: The future of high performance database systems. *Commun. ACM 35*, 6, 85–98.

FAN, W., CONG, G., AND BOHANNON, P. 2007. Querying XML with update syntax. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 293–304.

FLORESCU, D. AND KOSSMANN, D. 1999. Storing and querying XML data using an RDMBS. *IEEE Data Eng. Bull. 22*, 27–34.

GALAX. http://galax.sourceforge.net/.

GANESAN, P., BAWA, M., AND GARCIA-MOLINA, H. 2004. Online balancing of range-partitioned data with applications to Peer-to-Peer systems. In *Proceedings of the International Conference on Very Large Databases*. 444–455.

GODFREY, P. AND GRYZ, J. 2000. A strategy for partial evaluation of views. In *Proceedings of the International Joint Conference on Intelligent Information Systems*. 337–349.

GOTTLOB, G., KOCH, C., AND PICHLER, R. 2002. Efficient algorithms for processing XPath queries. In *Proceedings of the International Conference on Very Large Databases*. 95–106.

GUPTA, A., SAGIV, Y., ULLMAN, J. D., AND WIDOM, J. 1994. Constraint checking with partial information. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*. 45–55.

HALEVY, A. Y., IVES, Z. G., MADHAVAN, J., MORK, P., SUCIU, D., AND TATARINOV, I. 2004. The Piazza peer data management system. *IEEE Trans. Knowl. Data Engin. 16*, 7, 787–798.

HSIAO, H. AND DEWITT, D. J. 1990. Chained declustering: A new availability strategy for multiprocessor database machines. In *Proceedings of the International Conference on Data Engineering*. 456–465.

HUCK, G., MACHERIUS, I., AND FANKHAUSER, P. 1999. PDOM: Lightweight persistency support for the document object model. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*. 107–118.

IVES, Z. G., HALEVY, A. Y., AND WELD, D. S. 2002. An XML query engine for network-bound data. *VLDB J. 11*, 4, 380–402.

JAGADISH, H. V., LAKSHMANAN, L. V. S., MILO, T., SRIVASTAVA, D., AND VISTA, D. 1999. Querying network directories. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 133–144.

JAGADISH, H. V., OOI, B. C., AND VU, Q. H. 2005. Baton: A balanced tree structure for peer-to-peer networks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 661–672.

JONES, N. D. 1996. An introduction to partial evaluation. *ACM Comput. Surv. 28*, 3.

KANNE, C., BRANTNER, M., AND MOERKOTTE, G. 2005. Cost-sensitive reordering of navigational primitives. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 742–753.

KANNE, C.-C. AND MOERKOTTE, G. 2006a. The importance of sibling clustering for efficient bulkload of XML document trees. *IBM Syst. J. 45*, 2, 321–334.

KANNE, C.-C. AND MOERKOTTE, G. 2006b. A linear time algorithm for optimal tree sibling partitioning and approximation algorithms in Natix. In *Proceedings of the International Conference on Very Large Databases*. 91–102.

KAUSHIK, R., BOHANNON, P., NAUGHTON, J. F., AND KORTH, H. F. 2002. Covering indexes for branching path queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 133–144.

KLING, P., ÖZSU, M. T., AND DAUDJEE, K. 2010. Generating efficient execution plans for vertically partitioned XML databases. *Proc. VLDB Endow. 4*, 1, 1–11.

KOCH, C. 2003. Efficient processing of expressive node-selecting queries on XML data in secondary storage: A tree automata-based approach. In *Proceedings of the International Conference on Very Large Databases*. 249–260.

KOSSMAN, D. 2000. The state of the art in distributed query processing. *ACM Comput. Surv. 32*, 4, 422–469.

KUNDU, S. AND MISRA, J. 1977. A linear tree partitioning algonthm. *SIAM J. Comput. 6*, 1, 151–154.

LU, J., LING, T. W., CHAN, C. Y., AND CHEN, T. 2005. From region encoding to extended dewey: On efficient processing of XML twig pattern matching. In *Proceedings of the International Conference on Very Large Databases*. 193–204.

LUKES, J. A. 1974. Eficient algorithm for the partitioning of trees. *IBM J. Resear. Dev. 18*, 3, 217–224.

MARIAN, A. AND SIMÉON, J. 2003. Projecting XML documents. In *Proceedings of the International Conference on Very Large Databases*. 213–224.

MILO, T. AND SUCIU, D. 1999. Index structures for path expressions. In *Proceedings of the International Conference on Database Theory*. 277–295.

MONETDB. http://monetdb.cwi.nl/projects/monetdb/xquery /index.html.

PAL, S., CSERI, I., SCHALLER, G., SEELIGER, O., GIAKOUMAKIS, L., AND ZOLOTOV, V. 2004. Indexing XML data stored in a relational database. In *Proceedings of the International Conference on Very Large Databases*. 1146–1157.

PAPADIMOS, V. AND MAIER, D. 2002. Distributed queries without distributed state. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*. 95–100.

RAMANAN, P. 2002. Efficient algorithms for minimizing tree pattern queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 299–309.

SAXON. http://saxon.sourceforge.net/.

SCHMIDT, A., KERSTEN, M. L., WINDHOUWER, M., AND WAAS, F. 2000. Efficient relational storage and retrieval of XML documents. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*. 137–150.

SCHMIDT, A., WAAS, F., KERSTEN, M., CAREY, M., MANOLESCU, I., AND BUSSE, R. 2002. XMark: A benchmark for XML data management. In *Proceedings of the International Conference on Very Large Databases*. 974–985.

SMITH, M. AND HOWES, T. A. 1997. *LDAP: Programming Directory-Enabled Apps*. Sams.

SUCIU, D. 2002. Distributed query evaluation on semistructured data. *ACM Trans Datab. Syst. 27*, 1, 1–62.

TOMASIC, A., RASCHID, L., AND VALDURIEZ, P. 1996. Scaling heterogeneous databases and the design of Disco. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*. 449–457.

ZHANG, Y. AND BONCZ, P. A. 2007. XRPC: Interoperable and efficient distributed XQuery. In *Proceedings of the International Conference on Very Large Databases*. 99–110.

ZHANG, C., NAUGHTON, J. F., DEWITT, D. J., LUO, Q., AND LOHMAN, G. M. 2001. On supporting containment queries in relational database management systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 425–436.

ZHANG, N., KACHOLIA, V., AND OZSU, M. T. 2004. A succinct physical storage scheme for efficient evaluation of path queries in XML. In *Proceedings of the International Conference on Data Engineering*. 54.

ZHANG, Y., TANG, N., AND BONCZ, P. A. 2009. Efficient distribution of full-fledged XQuery. In *Proceedings of the International Conference on Data Engineering*. 565–576.