Using Partial Evaluation in Distributed Query Evaluation

Using Partial Evaluation in Distributed Query Evaluation

Peter Buneman Gao Cong
University of Edinburgh

Wenfei Fan [*]
University of Edinburgh &
Bell Laboratories

Anastasios Kementsietsidis
University of Edinburgh

{opb, gcong, wenfei, akements}@inf.ed.ac.uk

ABSTRACT

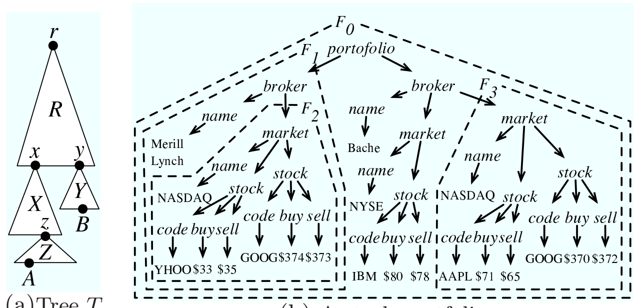
A basic idea in parallel query processing is that one is prepared to do more computation than strictly necessary at individual sites in order to reduce the elapsed time, the network traffic, or both in the evaluation of the query. We develop this idea for the evaluation of boolean XPath queries over a tree that is fragmented, both horizontally and vertically over a number of sites. The key idea is to send the whole query to each site which *partially evaluates*, in parallel, the query and sends the results as compact boolean functions to a coordinator which combines these to obtain the result. This approach has several advantages. First, each site is visited only once, even if several fragments of the tree are stored at that site. Second, no prior constraints on how the tree is decomposed are needed, nor is any structural information about the tree required, such as a DTD. Third, there is a satisfactory bound on the total computation performed on all sites and on the total network traffic. We also develop a simple incremental maintenance algorithm that requires communication only with the sites at which changes have taken place; moreover the network traffic depends neither on the data nor on the update. 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.

1. INTRODUCTION

Partial evaluation (*aka.* program specialization [17]) 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 depends only on s, and generates a partial answer, *i.e.,* a (residual) function f' that depends on the as yet unavailable input d.

Partial evaluation has been proven useful in a variety of areas including compiler generation, code optimization and dataflow evaluation (see [17] 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 an important fragment of XPath, Boolean XPath queries, which are commonly used, for example, in publish-subscribe systems [2] and LDAP directories [23].

[*]Supported in part by EPSRC GR/S63205/01, GR/T27433/01 and BBSRC BB/D006473/1.



(a)Tree T (b) A stock portfolio

Figure 1: Two fragmented XML trees

As an example, consider the simple Boolean XPath query $Q = [//A \wedge //B]$ over the XML tree T depicted in Fig. 1(a). We defer the discussion of the physical representation of T, but assume that T consists of four subtrees, or *fragments*, R, X, Y and Z, and that nodes with tags A or B only occur in fragments Z and Y as shown in Fig. 1(a). A sophisticated algorithm would do a single depth-first traversal of T computing $[//A]$ and $[//B]$ simultaneously [18], and visit each node exactly once, something we cannot expect to improve upon.

However, our recursive traversal visits fragments R, X, Z, X, R, Y, R in that order; it makes *three visits* to fragment R and *two visits* to fragment X. Could we devise an algorithm that makes *one visit* to each fragment? Observe that the evaluation of the query Q can be readily treated as the Boolean function $Q(R, X, Y, Z)$, *i.e.,* one whose value depends on all four fragments of the tree T. What does this function look like? Suppose that r_A, x_A, y_A and z_A are variables denoting the results of $[//A]$ respectively evaluated at the roots r of fragment R, x of fragment X, y of fragment Y and z of fragment Z, and that r_B, x_B, y_B, z_B are defined similarly. Then, it is not hard to see that:
$Q(R, X, Y, Z) = (r_A \vee x_A \vee y_A \vee z_A) \wedge (r_B \vee x_B \vee y_B \vee z_B)$
We can compute the results of $[//A]$ and $[//B]$ (the values of the variables) independently, and in *parallel*, in each fragment by accessing it *only once*. Each fragment returns the values of the corresponding variables and each time we

receive such values we use them to compute a partial answer for Q. Given that nodes with tags A and B only occur in fragments Z and Y, the execution of Q returns $(z_A, z_B) = (1, 0)$ on fragment Z, $(y_A, y_B) = (0, 1)$ on Y, $(x_A, x_B) = (z_A, z_B)$ on X and $(r_A, r_B) = (x_A \vee y_A, x_B \vee x_B)$ on R. Note that the process makes no assumption about the order in which it receives the values of the variables. A partial answer is computed each time values are received from some fragment. One order might compute the answer faster than some others but except this, order is of no consequence. Another remark is that some of the returned values are truth values while others are Boolean *expressions*. Irrespectively of the order and type of returned values, the process uses the returned values from all fragments to compute the answer to Q which, in this case, is *true*. This is a simple example of *partial evaluation*.

There are a number of database scenarios in which partial evaluation could be an effective optimization technique. For example, as in PDOM [15], suppose a large XML tree is stored in secondary storage and it is split into fragments. In this setting, the partial evaluation approach can save us the cost of two swaps of fragment R and one swap of fragment X. The benefit is already evident even though there is no parallelism involved in this example.

A more interesting scenario involves fragmented XML trees that are geographically or administratively distributed over the Internet [4], a setting commonly found in, *e.g.,* e-commerce, Web services, or while managing large-scale network directories [16]. More specifically, let us consider the XML tree shown in Fig. 1(b), which represents a person's stock *portfolio*. The person trades stocks through various *brokers* in possibly overlapping *markets*. For each *stock*, the tree stores its *code*, the price paid by the person to *buy* the stock, and the price at which the person can (currently) *sell* the stock. The same stock might be traded through a different broker at different periods of time and for a different price. For example, the GOOG stock is purchased both through *Merill Lynch* and through *Bache*. Although conceptually this is a single XML tree, in reality it is inherently distributed over the Internet. The figure uses dashed lines to show one possible fragmentation. For example, fragment F_0 includes the root of the tree and all the stock data from broker *Bache* in the NYSE market. This fragment might be stored locally in the persons' desktop. Broker *Merill Lynch* might require that all trade data are accessed through its own servers and thus fragment F_1 is stored there. In turn, the NASDAQ market might require that all its own data are only remotely accessed and only through recognized brokers. Therefore fragments F_2 and F_3 are both stored in its own servers. The NYSE market imposes no such restrictions and so its trade data can be stored locally by *Bache* (and *Merill Lynch*, although not shown in the figure). Notice that we make no assumptions about the size of fragments, their storage location, and the number of fragments assigned to each location. In a variety of applications, we have *no control* over these parameters and their values are imposed on us by the environment.

Assume that the portfolio owner wants to know whether the GOOG stock reaches a selling price of \$376. To do so, she must execute the boolean query $Q = [//stock[code = \text{“GOOG''} \wedge sell = 376]]$. There are two popular alternatives to execute such a query. The first alternative requires for the different sources to create a *stream* of data to the user and the query is executed over the received stream. There are two main concerns with this approach: (a) A large part of the tree that is later deemed irrelevant to the query, including the subtree for the NYSE market and the information for the YHOO and AAPL stocks, needs to be sent to the user, causing increased network traffic. The user might want to execute the query and be notified *on-the-go*, using a mobile device, like a cell phone. Streaming large data sets to the cell phone is particularly unrealistic. (b) Business or personal data are typically kept local at trusted sites, and are not shipped to other sites for security or privacy concerns. In industry and research, similar concerns have generated increasing interest, and a preemergent shift, towards shipping the processing (queries) to the data, instead of shipping (streaming) the data to the processing [11].

The second alternative is to use a publish-subscribe system. Assuming that the tree in Fig. 1(b) is part of such a system, the user needs to issue query Q over the system and whenever the query predicate is satisfied, the user is notified. Publish-subscribe systems are more in-line with moving the processing to the data. However, any publish-subscribe system implementing the optimal centralized algorithm in XPath query processing [18] would require a single depth-first traversal of the document tree visiting, in our example, twice the NASDAQ server.

In response to these, we propose partial evaluation as a practical query processing technique since it ensures that each remote site is visited only *once*. Moreover, while a depth-first traversal serializes the processing of the different fragments, partial evaluation can speed-up query processing since fragments at different sites are processed *in parallel*. Unlike streaming, partial evaluation executes the queries where the data reside, thus minimizing network traffic and facilitating the execution of (complex) queries over devices with limited bandwidth.

Our first contribution consists of several algorithms for evaluating Boolean XPath queries over a fragmented tree, with the following performance guarantees. (a) Each site holding a fragment is visited *only once*. (b) The total network traffic 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 fragmented, what the sizes of these fragments are, or how they are assigned to sites.

Our second contribution is a simple incremental maintenance algorithm for Boolean XPath views. Cached views and their incremental maintenance are important for efficient query answering [13, 25]. Our algorithm has the following features: (a) The computation is *localized* so that only the site where the updates take place is visited, and reevaluation is only conducted on the updated fragment; no other sites or fragments are visited. (b) The total network traffic depends on *neither the data nor the update*.

Our third contribution is an experimental study evaluating our algorithms versus optimal centralized approaches. Our experimental results show that our technique outperforms, by a substantial factor in all practical cases, existing centralized algorithms for evaluating XPath queries.

These yield the first Boolean XPath evaluation and incremental maintenance algorithms with *performance guarantee* in distributed systems. The technique generalizes to data selection XPath queries, as be seen in Section 8; we consider Boolean queries in this paper to focus on the main idea of partial evaluation and to simplify the discussion. As mentioned earlier, the partial evaluation technique is more general than this application. In fact, other proposals for distributed query evaluation can be seen as related to partial evaluation. We shall discuss these in Section 7.

Organization. Section 2 discusses XML tree fragmentation and reviews Boolean XPath. Section 3 presents our main algorithms for processing Boolean XPath queries based on partial evaluation. Variations of the basic algorithms are given in Section 4. Section 5 develops our incremental maintenance algorithm for XPath views. An experimental study is provided in Section 6, followed by related work in Section 7 and conclusions in Section 8.

2. BACKGROUND

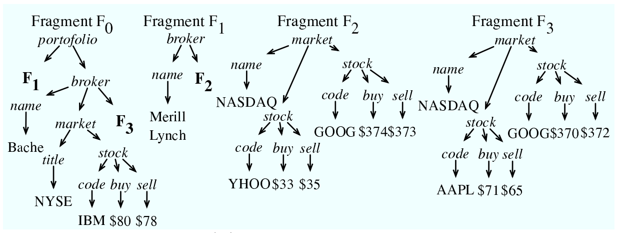
We next discuss the distribution of XML documents and present the class of XPath queries studied in this paper.

2.1 XML Tree Fragmentation

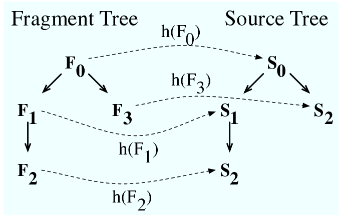
We allow an XML tree T to be decomposed into a collection $\mathcal{F}$ of disjoint trees, or *fragments* F_i, which are distributed to and stored in different sites. A tree may be fragmented for administrative reasons (*e.g.,* different sites are inherently responsible for maintaining different parts of the tree), or for efficiency [4] (*e.g.,* the tree might be too big to store in a single site). We *do not impose any constraints* on the fragmentation: we allow for an arbitrary "nesting" of fragments. Each fragment 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: this is determined by the system. Hence our fragmentation setting is the most generic possible, making our solutions applicable in almost every conceivable setting. For instance, Fig. 2(a) shows the decomposition of the tree in Fig. 1(b) in four fragments. To the left of Fig. 2(b) we summarize this decomposition in a structure called the *fragment tree*. Note that the fragment F_1 is itself fragmented.

Next, we define a few useful notation about fragments. The fragment that contains the root of the tree T, *i.e.,* the fragment at the root of the fragment tree, is called the *root* fragment. In Fig. 2, this is fragment F_0. Given two fragments F_j and F_k, we say that F_k is a *sub-fragment* of F_j if F_k is a child of F_j in the fragment tree. If F_k is a sub-fragment 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. In Fig. 2(b), fragment F_3 is a sub-fragment of F_0 and in the tree of Fig. 1(b) there is an edge between the *broker* node of F_0 and the root node *market* of F_3.

We maintain the relationship between a fragment and its sub-fragments to preserve the structure of the original tree T. To do this, we add to the v node of fragment F_j a *virtual* child node with label F_k. While traversing fragment F_j, we know that if we reach the virtual node F_k, we need to "jump" to fragment F_k in order to continue the traversal. In Fig. 2(a), fragment F_0 has the virtual node F_1 which in turn has the virtual node F_2. Finally, we refer to a fragment that has no sub-fragments as a *leaf* fragment. In Fig. 2(b), both fragments F_2 and F_3 are leaf fragments.



(a) Four fragments



(b) The Fragment tree and the source tree

Figure 2: Tree fragmentation

As fragments are distributed among sites, it is important to keep track of where the different fragments reside. We assume that there is a mapping function h, which encodes the assignment of fragments to sites (sources). To the right of Fig. 2(b) we show a structure called the *source tree*, which is induced from the fragment tree and the function h. The source tree of a tree T, denoted by $\mathcal{S}_T$, shows the names of sites where the fragments of T are stored. From the source tree we can see that *both* fragments F_2 and F_3 are stored in the *same* site S_2, the NASDAQ site.

The source tree $\mathcal{S}_T$ is the *only structure* that our XPath evaluation and incremental maintenance algorithms require. No other information about either the fragmentation or the distribution of the XML tree is needed. The fragment tree was introduced to illustrate fragmentation, and is neither maintained nor used by our system.

2.2 Boolean XPath

We consider a class of Boolean XPath queries, denoted by $\mathcal{X}_{\rm BL}$. A query $[q]$ in $\mathcal{X}_{\rm BL}$ is syntactically defined as follows:

$$\begin{array}{lll} q & := & p \;\mid\; p/\mathit{text}() = \mathit{str} \;\mid\; \mathit{label}() = A \;\mid\; \neg q \;\mid\; q \wedge q \;\mid\; q \vee q, \\ p & := & \epsilon \;\mid\; A \;\mid\; * \;\mid\; p//p \;\mid\; p/p \;\mid\; p[q], \end{array}$$

where *str* is a string constant, A is a label (tag), $\neg, \wedge, \vee$ are the Boolean negation, conjunction and disjunction operators, respectively; p is a *path expression* defined in terms of the empty path ϵ (*self*), label A, wildcard $*$, the *descendant-or-self-axis* '//', child '/', and *qualifier* $[q]$. For //, we abbreviate $p_1/$ // as $p_1//$ and // $/p_2$ as $//p_2$.

For example, $[//\mathit{broker}[//\mathit{stock}/\mathit{code}/\mathit{text}() = \text{"GOOG"} \wedge \neg (//\mathit{stock}/\mathit{code}/\mathit{text}() = \text{"YHOO"})]]$ is a query in $\mathcal{X}_{\rm BL}$.

Note that path expressions p in $\mathcal{X}_{\rm BL}$ subsume tree pattern queries and beyond, which are commonly studied in the literature. As mentioned earlier, queries in $\mathcal{X}_{\rm BL}$ are widely used in, *e.g.,* XML data dissemination for content-based filtering and routing of XML documents. Although we consider Boolean queries, the technique generalizes to a larger class of queries, which are discussed in the conclusions.

At a *context node* v in an XML tree T, the evaluation of a query $[q]$ yields a truth value, denoted by $\mathsf{val}(q, v)$, indicating whether or not q is satisfied at v. Specifically, (a) when q is a path p, $\mathsf{val}(q, v)$ is *true* iff there exists a node reachable from v via p; (b) when q is $p/text() = str$, $\mathsf{val}(q, v)$ is *true* iff there is a node u reached from v via p such that u carries text value str; similarly when q is $label() = A$; (c) when q is $q_1 \wedge q_2$, $\mathsf{val}(q, v)$ is *true* iff both $\mathsf{val}(q_1, v)$ and $\mathsf{val}(q_2, v)$ are *true*; similarly when q is $q_1 \vee q_2$ or $\neg q_1$.

On a *centralized* XML tree T, *i.e.,* when T is not decomposed and distributed, $\mathsf{val}(q, r)$ can be computed in $O(|T|\ |q|)$ time [10], where r is the root of T.

To simplify the presentation we introduce two notation, which will be used in our algorithms given in later sections.

First, we rewrite each path p in an $\mathcal{X}_{\rm BL}$ query $[q]$ to a normal form $\beta_1/\ldots/\beta_n$, where β_i is one of ϵ, $*$, $//$ or $\epsilon[q']$. This normalization can be achieved by using a linear-time function $\mathsf{normalize}(q)$, given inductively as follows:

$$
\begin{array}{ll}
\mathsf{normalize}(\epsilon) & = \epsilon\text{; similarly for `}*\text{', `}//\text{' and } label() = A; \\
\mathsf{normalize}(A) & = */\epsilon[label() = A]; \\
\mathsf{normalize}(p_1/p_2) & = \mathsf{normalize}(p_1)/\mathsf{normalize}(p_2); \\
\mathsf{normalize}(p[q']) & = \mathsf{normalize}(p)/\epsilon[\mathsf{normalize}(q')]; \\
\mathsf{normalize}(q_1 \wedge q_2) & = \mathsf{normalize}(q_1) \wedge \mathsf{normalize}(q_2); \\
 & \quad \text{similarly for } q_1 \vee q_2 \text{ and } \neg q_1; \\
\mathsf{normalize}(p/\text{text}() = \text{`}str\text{'}) & = \mathsf{normalize}(p)[\text{text}() = \text{`}str\text{'}]; \\
\mathsf{normalize}(\epsilon[q_1]/\ldots/\epsilon[q_n]) = & \\
\quad \epsilon[\mathsf{normalize}(q_1) \wedge \mathsf{normalize}(q_2) \wedge \ldots \wedge \mathsf{normalize}(q_n)]; &
\end{array}
$$

where the last rule is to combine a sequence of ϵ's into one. In the sequel we consider $\mathcal{X}_{\rm BL}$ queries in the normal form.

Second, we use $\mathsf{QList}(q)$ to denote the list of all sub-queries of q. Intuitively, q_1 is a sub-query of q if the parse tree of q_1 is a subtree of that of q. We sort $\mathsf{QList}(q)$ in a topological order such that for any sub-queries q_1, q_2 of q, if q_1 is a sub-query of q_2 then q_1 precedes q_2 in $\mathsf{QList}(q)$.

Example 2.1: Consider the $\mathcal{X}_{\rm BL}$ query $[q]$, where q is $//stock[code/text() = \text{"YHOO"}]$, then

$$
\begin{array}{ll}
\mathsf{normalize}([q]) = \epsilon[//\epsilon[label() = stock \wedge & \\
\qquad */\epsilon[label() = code \wedge text() = \text{"YHOO"}]]], & \\
\mathsf{QList}([q]) \quad = [q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}], \text{ where} & \\
q_1 = label() = code, & q_2 = (text() = \text{"YHOO"}), \\
q_3 = q_1 \wedge q_2, & q_4 = \epsilon[q_3], \\
q_5 = */\epsilon[q_4], & q_6 = (label() = stock), \\
q_7 = q_5 \wedge q_6, & q_8 = \epsilon[q_7], \\
q_9 = //\epsilon[q_8], & q_{10} = \epsilon[q_9]
\end{array}
$$

□

Remark. Observe that both $\mathsf{normalize}(q)$ and $\mathsf{QList}(q)$ can be computed in $O(|q|)$ time. As a result, the *total size* of sub-queries in the list $\mathsf{QList}(q)$ is *bounded by* $O(|q|)$. Furthermore, for any XPath evaluation algorithm, to evaluate q it is necessary to evaluate sub-queries in $\mathsf{QList}(q)$.

3. DISTRIBUTED QUERY EVALUATION

Consider an $\mathcal{X}_{\rm BL}$ query q submitted to a site S, hereafter referred to as the *coordinating* site. The query is to be evaluated at the root of a fragmented and distributed XML tree T. A naïve evaluation is to collect all the fragments of tree T identified by the source tree $\mathcal{S}_T$ at the coordinating site, and use a *centralized* algorithm, *e.g.,* the algorithm of [10]. 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 need to be sent over the network, *each* time a query needs to be executed. In addition, since the coordinating site must store these fragments during the evaluation of q, the benefits gained by our ability to distribute 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 source 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 source 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 sub-fragments F_k in site S_l. For example, consider the fragment and source trees in Fig. 2(b). 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_3 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 sub-fragments 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_3. For each of these visits, site S_2 has to exchange a number of messages, 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, in any practical setting, is determined by *the size of the query* rather than the XML tree.

3.1 The ParBoX Algorithm

The algorithm is initiated at the coordinating site which, without loss of generality, we assume to be the site storing the root fragment of the tree T over which the $\mathcal{X}_{\rm BL}$ query q is evaluated. The algorithm consists of three stages:

Stage 1: Initially (lines 1-2 of Procedure ParBoX in Fig. 3(a)), the coordinating site uses the source tree $\mathcal{S}_T$ to identify which other sites hold at least one fragment of tree T. In our example, coordinating site S_0 uses source tree in Fig. 2(b) to identify sites S_1 and S_2.

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, Fig. 3(b)). 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 (the second stage), and (b) how to assemble the partial answers to obtain the answer to query q (the third stage).

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 sub-queries $\mathsf{QList}(q)$ of query q, where $\mathsf{QList}(q)$ is described in Section 2.2. This computation requires the (already computed) values of the $\mathsf{QList}(q)$ sub-queries at the children of v. At the end of the traversal, the answer to query q is computed by the values of the $\mathsf{QList}(q)$ queries at the root of the tree. Specifically, the answer to q is the value of the last query in $\mathsf{QList}(q)$.

Consider now Fig. 2(a) which shows the fragments of the XML tree in Fig. 1(b). 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, *i.e.,* they are pointers to other fragments which 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 two virtual leaves, one for fragment F_1 and one for F_3. In accordance to the strategy given above, 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 $\mathsf{QList}(q)$ sub-queries at the virtual nodes? The values of the $\mathsf{QList}(q)$ sub-queries 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 $\mathsf{QList}(q)$ sub-query at each virtual node. Using these variables, the bottom-up evaluation procedure is given in Fig. 3(b). Procedure bottomUp considers the root of a fragment F_j and a list q_L of sub-queries which is essentially the $\mathsf{QList}(q)$ 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 encountered, the procedure computes the "values" of q_L at v and stores the results of the computation in a vector V_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. To cope with this, we save these values (lines 3-5) by maintaining only two additional vectors, namely vectors CV_v and DV_v, that are of the same size as vector V_v. Intuitively, for each sub-query q' in q_L, $CV_v(q')$ is true if and only if there exists some child u of v such that $V_u(q')$ is true, and similarly, $DV_v(q')$ is true if and only if either $V_v(q')$ is true or there exists some descendant w of v such that $V_w(q')$ 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

```
Procedure ParBoX
Input: An $\mathcal{X}_{\rm BL}$ query $q$ and a fragmented, distributed tree $T$
Output: The boolean value $ans$ of $q$ over $T$
1. $q_L$ := QList($q$);
2. $\mathcal{S}_T$ := retrieve the source tree of tree $T$;
3. for each site $S_i$ in the source tree $\mathcal{S}_T$ do
4.     execute($S_i$, evalQual, $q_L$);
5.     for each fragment $F_j$ stored in $S_i$ do
6.         annotate $\mathcal{S}_T$ with $(V_{F_j}, CV_{F_j}, DV_{F_j})$;
7. $ans$ := evalST($\mathcal{S}_T$);
```

(a) ParBoX algorithm executed at coordinating site

```
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.     $(V_{F_j}, CV_{F_j}, DV_{F_j})$ = bottomUp($root(F_j)$, $q_L$);
3.     send $(V_{F_j}, CV_{F_j}, DV_{F_j})$ to the coordinating site;

Procedure bottomUp
Input: A node $v$ and a list $q_L$ of (sub-)queries
Output: Vectors $V_v$, $CV_v$ and $DV_v$ of formulas for node $v$
1.  for each child $w$ of $v$ do
2.      $(V_w, CV_w, DV_w)$ := bottomUp($w$, $q_L$);
3.      for each query $q_i$ in $q_L$ do
4.          $CV_v(q_i)$ := compFm($CV_v(q_i), V_w(q_i), OR$);
5.          $DV_v(q_i)$ := compFm($DV_v(q_i), DV_w(q_i), OR$);
6.  for each query $q_i$ in $q_L$ from left to right do
7.      case $q_i$ of
8.      (c0) $\epsilon$: $V_v(q_i) := 1$;
9.      (c1) $label() = l$: $V_v(q_i)$ := $compareString(label(), l)$;
10.     (c2) $text() = str$: $V_v(q_i)$ := $compareString(text(), str)$;
11.     (c3) $*/q_j$: $V_v(q_i) := CV_v(q_j)$;
12.     (c4) $\epsilon[q_j]/q_k$: $V_v(q_i)$ := compFm($V_v(q_j), V_v(q_k), AND$);
13.     (c5) $//q_j$: $V_v(q_i) := DV_v(q_j)$;
14.     (c6) $q_j \vee q_k$: $V_v(q_i)$ := compFm($V_v(q_j), V_v(q_k), OR$);
15.     (c7) $q_j \wedge q_k$: $V_v(q_i)$ := compFm($V_v(q_j), V_v(q_k), AND$);
16.     (c8) $\neg q_j$: $V_v(q_i)$ := compFm($V_v(q_j), NULL, NEG$);
17.     $DV_v(q_i)$ := compFm($V_v(q_i), DV_v(q_i), OR$);

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 $op = AND$ then $ans := f_1 \vee f_2$;
```

(b) ParBoX algorithm executed at participating site

Figure 3: The ParBoX Algorithm

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. As a consequence of recursive evaluation, this value has already been accumulated in $CV_v(q_j)$. Similarly, when q_i is $//q_j$ (line 13), the value of q_i at node v is the disjunction of $V_v(q_j)$ and $DV_w(q_j)$'s for the children w of v, which have again been computed due to the bottom-up processing order following the list q_L of sub-queries. 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 our 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: Recall query q from Example 2.1. Evaluating the values of the (sub-)queries in $q_L = \mathsf{QList}(q)$ (given in Example 2.1) for the nodes in fragment F_1 results in the following V_v vectors:

- $V_{name} =< 0,0,0,0,0,0,0,0,0,0 >$
- $V_{F_2} =< x_1, x_2, x_3 = x_1 \wedge x_2, x_4 = x_3, x_5 = cx_4,$
  $x_6, x_7 = x_5 \wedge x_6, x_8 = x_7, x_9 = dx_8, x_{10} = x_9 >$
- $V_{broker} =< 0,0,0,0,x_4,0,0,0,dx_8,dx_8 >$

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 V_{F_2}, CV_{F_2} and DV_{F_2} vectors, respectively, of virtual node F_2. Note that for each (sub-)query of node 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 $(V_{F_j}, CV_{F_j}, DV_{F_j})$ 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 (V_v, CV_v, DV_v) and one for its children (V_w, CV_w, DV_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:

- $V_{F_0} =< 0,0,0,0,y_4,0,0,0,dy_8 \vee dz_8, dy_8 \vee dz_8 >$
  $CV_{F_0} =< y_1, y_2, y_3, y_4, y_5 \vee z_4, y_6, y_7, y_8,$
  $y_9 \vee dz_8, y_{10} \vee dz_8 >$
  $DV_{F_0} =< dy_1 \vee dz_1, dy_2 \vee dz_2, dy_3 \vee dz_3, dy_4 \vee dz_4,$
  $dy_5 \vee dz_5 \vee z_4 \vee y_4, 1, dy_7 \vee dz_7, dy_8 \vee dz_8,$
  $dy_8 \vee dz_8 \vee dy_9 \vee dz_9, dy_8 \vee dz_8 \vee dy_{10} \vee dz_{10} >$
- $V_{F_1} =< 0,0,0,0,x_4,0,0,0,dx_8,dx_8 >$
  $CV_{F_1} =< 0,x_2,x_3,x_4,x_5,x_6,x_7,x_8,x_9,x_{10} >$
  $DV_{F_1} =< 0,dx_2,dx_3,dx_4,x_4 \vee dx_5,dx_6,dx_7,dx_8,$
  $dx_8 \vee dx_9, dx_8 \vee dx_{10} >$
- $V_{F_2} =< 0,0,0,0,0,0,0,0,1,1 >$
  $CV_{F_2} =< 0,0,0,0,1,1,1,1,1,1 >$
  $DV_{F_2} =< 1,1,1,1,1,1,1,1,1,1 >$
- $V_{F_3} =< 0,0,0,0,0,0,0,0,0,0 >$
  $CV_{F_3} =< 0,0,0,0,0,1,0,0,0,0 >$
  $DV_{F_3} =< 1,0,0,0,0,1,0,0,0,0 >$

In the triplets, variables x_i, cx_i and dx_i for $1 \le i \le 10$ are used in fragment F_1 to represent the values of the virtual node F_2, while variables y_i, cy_i, dy_i and z_i, cz_i, dz_i are used in fragment F_0 to represent the values of the virtual nodes F_1 and F_3, 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 source tree, Procedure evalST (not shown due to the space constraint) needs a single bottom-up traversal of the source tree to solve the system of equations and find the answer to query q. Note that the vectors of leaf fragments in the source tree contain no variables. This is the case for both fragments F_2 and F_3. During the bottom-up traversal of $\mathcal{S}_T$, Procedure 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 $V_{F_{root}}(q_{last})$, where F_{root} is the root fragment and q_{last} is the last query in the q_L list.

Example 3.3: Consider the source tree in Fig. 2(b) and the vectors of the fragments from our previous example. Then, the answer to query q is the value of the last query in V_{F_0}, that is, $q = dy_8 \vee dz_8$. A bottom-up evaluation of Procedure evalST uses DV_{F_2} to unify dx_8 to 1; DV_{F_1} to unify dy_8 to dx_8; and DV_{F_3} to unify dz_8 to 0. Therefore, $q = 1$ and the query q evaluates to *true.* □

3.2 Analysis

For the complexity of the Algorithm ParBoX, we consider its communication cost as well as the *total* and *parallel* computation costs for evaluating a query q on a fragmented 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 notation: $\mathcal{F}$ 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 sub-fragments of fragment F_j. We use $\mathsf{card}(\mathcal{X})$ to denote the cardinality of a set $\mathcal{X}$.

Total network traffic. Observe that each site appearing in the source 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, *i.e.,* $\mathsf{card}(\mathcal{F}_j)$, due to the variables introduced by these virtual nodes. In the worst case, the size of the entry is in $O(|\mathcal{F}_j|)$. Therefore, the communication cost for each fragment F_j is $O(|q|\mathsf{card}(\mathcal{F}_j))$ and the overall communica-

Algorithm	Visits		Computation	Communication
NaiveCentralized	1		$O(\|q\|\|T\|)$	$O(\|T\|)$
NaiveDistributed	$\mathsf{card}(\mathcal{F}_{S_i})$		$O(\|q\|\|T\|)$	$O(\|q\|\mathsf{card}(\mathcal{F}))$
ParBoX	1	T	$O(\|q\|(\|T\| + \mathsf{card}(\mathcal{F})))$	$O(\|q\|\mathsf{card}(\mathcal{F}))$
		P	$O(\|q\|(\max_{S_i}(\|F_{S_i}\|) + \mathsf{card}(\mathcal{F}))$	
Hybrid ParBoX	1	T	$O(\|q\|\|T\|)$	$O(\|T\|)$
		P	$O(\|q\|(\max_{S_i}(\|F_{S_i}\|) + \mathsf{card}(\mathcal{F}))$	
FullDistParBoX	$\mathsf{card}(\mathcal{F}_{S_i})$	T	$O(\|q\|(\|T\| + \mathsf{card}(\mathcal{F})))$	$O(\|q\|\mathsf{card}(\mathcal{F}))$
		P	$O(\|q\|(\max_{S_i}(\|F_{S_i}\|) + \mathsf{card}(\mathcal{F}))$	
LazyParBoX	$\mathsf{card}(\mathcal{F}_{S_i})$	T	$O(\|q\|(\|T\| + \mathsf{card}(\mathcal{F})))$	$O(\|q\|\mathsf{card}(\mathcal{F}))$
		P	$O(\|q\|\mathsf{card}(\mathcal{F}) \max_{\mathcal{F}}(\|F_i\|))$	

Figure 4: Summary of presented algorithms

tion cost of the algorithm is $O(|q|\Sigma_{j=1}^{\mathsf{card}(\mathcal{F})}\mathsf{card}(\mathcal{F}_j))$, that is, $O(|q|\mathsf{card}(\mathcal{F}))$ (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 which is of size $O(|q|\mathsf{card}(\mathcal{F}))$. Overall, the total amount of computation of Algorithm ParBoX is $O(|q|(|T| + \mathsf{card}(\mathcal{F})))$.

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 $O(|q|(\max_{S_i}(|F_{S_i}|) + \mathsf{card}(\mathcal{F})))$.

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, *i.e.*, $\mathsf{card}(\mathcal{F}) << |T|$. Thus, given a decomposition of a tree T to a set of fragments, our algorithm 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 our algorithm becomes $O(|q||T|)$, comparable to that of the best-known centralized algorithm [10, 18] 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. VARIATIONS OF ALGORITHM ParBoX

We next adapt Algorithm ParBoX to various settings.

Hybrid ParBoX. Although very unlikely to occur, it is instructive to study the pathological case in which each node v of our document tree T constitutes a separate fragment F and it is assigned to a different site. Then, we have that $\mathsf{card}(\mathcal{F}) = |T|$. Even in this pathological case, the computation cost of our algorithm is still optimal. However, the communication cost becomes $O(|q||T|)$, no longer independent of the tree T. Consider now the algorithm NaiveCentralized outlined in Section 3. The computation cost of the naïve al-

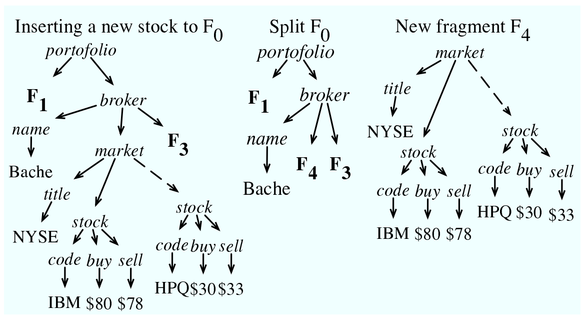


Figure 5: Updates on data and fragments

gorithm is still $O(|q||T|)$ but its communication cost is only $O(|T|)$. Therefore, for the pathological case considered, the naïve algorithm has less communication overhead than Algorithm ParBoX. This leads us to consider a *hybrid* algorithm that, depending on the decomposition of the input tree T, it behaves like Algorithm ParBoX for most decompositions but switches to the naïve algorithm for pathological decompositions. The tipping point in this switching of behaviour is determined by comparing $\mathsf{card}(\mathcal{F})$ and $\frac{|T|}{|q|}$. As long as $\mathsf{card}(\mathcal{F}) < \frac{|T|}{|q|}$, Algorithm ParBoX has less communication overhead than the naïve algorithm. The total computation cost of the hybrid algorithm is $O(|q||T|)$ while its communication cost is $O(|T|)$, in the worst case, and $O(|q|\mathsf{card}(\mathcal{F}))$ on average. The parallel computation cost of Hybrid ParBoX is the same as that of Algorithm ParBoX.

Full distribution of computation. When a large number of queries are submitted to the same coordinating site, the coordinating site might turn out to be a system bottleneck. As the coordinating site has to collect and process the partial answers of participating sites, it might be overwhelmed by both the amount of incoming traffic and the size of data to be processed. We address this issue in a new algorithm called FullDistParBoX by distributing the computation of the third phase of Algorithm ParBoX among all the participating sites. The first two phases of Algorithm FullDistParBoX are the same as those of ParBoX. During the third phase, Algorithm FullDistParBoX calls procedure evalDistrST instead of Procedure evalST.

In a nutshell, Procedure evalDistrST (omitted due to space constraints) assumes that each participating site holds a copy of the source tree $\mathcal{S}_T$. Given that the size of $\mathcal{S}_T$ is expected to be much smaller than the size of T, the assumption adds minimum storage overhead per site. The procedure proceeds in a bottom-up fashion in $\mathcal{S}_T$ by considering initially the sites that appear as leaves in $\mathcal{S}_T$. Consider such a site S and assume that it is responsible for a leaf fragment F. Site S sends the triplet of vectors corresponding to F to its parent site S' in $\mathcal{S}_T$. A non-leaf site, like site S', considers each local fragment F' and, after receiving the triplets of all sub-fragments of F', S' executes locally Procedure evalST using the received triplets along with the triplet for F'. Then, it sends the resulting triplet for F' to its parent site in $\mathcal{S}_T$. Note that S' and S still partially evaluate their local fragments in parallel in the second stage of the algorithm. Also note that no variables appear in the resulting triplet of vectors of F'. The process terminates when it reaches the site at the root of $\mathcal{S}_T$. Procedure evalDistrST has the same total/parallel computation and communication costs as Procedure evalST. Thus Algorithm FullDistParBoX is similar to Algorithm ParBoX. In practical terms we expect that the communication cost of the former algorithm is lower than that of the latter. Indeed, in the former, no variables are sent between sites since they are always unified, before any vector is sent. Although Algorithm FullDistParBoX removes the need for a coordinating site it has the drawback that a site might be visited multiple times, once for each time it appears in $\mathcal{S}_T$.

Lazy computation: Algorithm ParBoX is *eager* in that it requests all the sites to evaluate the queries in QList(q) over all their fragments. This approach maximizes parallelism but it does, in certain cases, result in unnecessary computation. To see why this is so, consider the following query, which checks whether there exists in our collection any broker with the name *"Merill Lynch"*:

[*/portofolio/broker/name* = *"Merill Lynch"*]

Note that although we do not need to compute the query on fragments F_2 and F_3, Algorithm ParBoX will do so. We can save some of this unnecessary computation by using a *lazy* strategy that evaluates the query in increasing depths of the site tree $\mathcal{S}_T$ until it obtains an answer to the query that does not depend on any fragments further down the source tree than the currently evaluated depth.

Algorithm LazyParBoX (also omitted) traverses the source tree $\mathcal{S}_T$ in pre-order. At the i^{th} *traversal step* of the traversal, the coordinating site identifies all the sites that hold fragments at depth i from the root of the source tree. For each of these sites, the coordinator requests the evaluation of Procedure evalQual for the corresponding fragments. The coordinating site collects the evaluated vectors for all these fragments and, along with the vectors collected from previous traversal steps, it calls Procedure evalST to compute the answer to the query. Only if no answer can be computed, due to variables that cannot be unified, the algorithm performs one more step. The total computation cost of Algorithm LazyParBoX is the same as that of ParBoX. However, in cases such as our last example, the algorithm behaves better than ParBoX. In our example, LazyParBoX does not evaluate the query over fragment F_2, since after one step the given query is evaluated to *true*. In terms of parallel computation, Algorithm LazyParBoX is worse than ParBoX since in each traversal step only one fragment is evaluated per site, and only fragments at the same level of the source tree are computed in parallel. Thus the parallel computation cost of the algorithm is $O(|q||\mathsf{card}(\mathcal{F}) \max_{\mathcal{F}}(|F_i|))$, where $\max_{\mathcal{F}}(|F_i|)$ denotes the size of the maximum fragment of tree T.

We summarize the discussion of these algorithms in Fig. 4, in which we list the number of times each site is visited, the total (T) and parallel (P) computation costs, and the communication costs. Recall that the first two naïve algorithms do not exploit parallelism and thus we only report their total computation costs. In Fig. 4 we use $\mathsf{card}(\mathcal{F}_{S_i})$ to denote the number of fragments that reside in site S_i.

5. INCREMENTAL VIEW MAINTENANCE

As remarked in Section 1, one often wants to cache the result of a query, treat it as a materialized view, and use it to answer possible future queries (see, *e.g.,* [13, 25]). When a new query is issued, the materialized views are used to provide part of, or the whole of, the answer to the query. With this comes the issue of view maintenance: when the source data is updated, the materialized views must be maintained so as to reflect the current source contents. An approach to maintaining views is by means of an incremental technique: given a query Q, a database I, a view $V = Q(I)$ and updates Δ_I on the source I, we compute update Δ_V on the view such that $V \oplus \Delta_V = Q(I \oplus \Delta_I)$. Incremental maintenance of views has proven effective in many applications (see, *e.g.,* [12, 26]), since small changes Δ_I to the source often inflict only small changes Δ_V to the view, and thus it is often more efficient to compute Δ_V rather than computing the view $Q(I \oplus \Delta_I)$ starting from scratch.

We next present a mechanism to support incremental view maintenance that is based on extensions of our XPath evaluation algorithms. Our incremental algorithms have the following salient features. (a) The cost of maintaining materialized views depends *neither on the size of the data nor on the size of the update.* (b) The recomputation is *localized* to the fragments where the updates occur.

Materialized view. A materialized view M of a query q over a tree T, denoted as $M(q,T)$, is a pair $(\mathcal{S}_T, ans)$, where $\mathcal{S}_T$ is the source tree of T and ans is the cached answer of the query q over T. We refer to the pair $(\mathcal{S}_T, ans)$ as the *state* of view M. A view is materialized at a site if the site maintains its state. Our maintenance algorithm imposes minimum overhead on the site, since only the query, the source tree and the answer need to be stored.

Update operations. We consider two classes of updates that can alter the state of a materialized view: the first alters the contents of the tree T and the second alters the fragmentation of T. For each class, there are two primitive operations, which are listed below. All operations are defined with respect to a fragment F_j of tree T.

(1) insNode(A, v): inserts in F_j a node labeled A as a child of node v. The operation returns the newly inserted node.

(2) delNode(v): deletes from F_j the node v.

(3) splitFragments(v): creates a new fragment F_k which is the subtree rooted at node v. The new fragment F_k is a sub-fragment of F_j and its subtree is replaced in F_j by a virtual node whose label is F_k.

(4) mergeFragments(v): merges fragment F_j with the sub-fragment that corresponds to the virtual node v. If v is not virtual, no action is taken.

Example 5.1: Consider fragment F_0 in Fig. 2(a). We can use a series of insNode operations to insert a new stock in the fragment, yielding the fragment in the left of Fig. 5. The new subtree is indicated by dotted lines. We can continue by applying operation splitFragments($market$) to the new fragment and get two fragments: a revised F_0 and a new fragment F_4 shown in Fig. 5. Note that the subtree rooted at the *market* node is replaced by a virtual node F_4. Fragment F_4 can now be assigned to another site, say, S_3. □

Since the first two operations concern the contents of a fragmented tree T, they only affect the ans part of the state of a view $M(q, T)$. Given a series of insertions and/or deletions in a fragment F_j, we extend Algorithm ParBoX to incrementally update ans; extensions to the variations of the Algorithm ParBoX are done similarly.

Algorithm outline. To incrementally update ans, it suffices to augment the state of $M(q, T)$ so that we also maintain the triplets $(V_{F_k}, CV_{F_k}, DV_{F_k})$, for each of the fragments F_k of T. After the series of insertions and/or deletions in fragment F_j, *only the site storing* F_j needs to call Procedure bottomUp and *only for fragment* F_j. The resulting triplet $(V_{F_j}^{new}, CV_{F_j}^{new}, DV_{F_j}^{new})$ is sent back to the site S storing the state of $M(q, T)$. The triplet is then compared with the one stored in S for the same fragment F_j. If they are identical, incremental evaluation terminates without changing the value of ans. Otherwise site S uses the new triplet, along with the triplets for the other fragments, in Procedure evalST to compute the new value of ans.

The total (and parallel) computation cost of the incremental algorithm is $O(|q|(|F_j| + \mathsf{card}(\mathcal{F})))$ while the communication cost is $O(|q|\mathsf{card}(\mathcal{F}_j))$, where $|F_j|$ is the size of the fragment F_j while $\mathsf{card}(\mathcal{F}_j)$ and $\mathsf{card}(\mathcal{F})$ are the number of sub-fragments of F_j and T, respectively. Observe that the communication cost is *independent* of both $|T|$ and the size of the updates. Furthermore, recomputation is localized to fragment F_j, in which updates take place.

Now consider splitFragments(v), which splits a fragment F_j into two fragments F_j^{new} and F_k. Obviously, the splitting does not affect the value of ans. However, both the source tree $\mathcal{S}_T$ and the corresponding fragment vector triplets must be updated. This update is local to the site S storing the state of M. The only other site involved in the process is site S' where fragment F_j used to reside. Site S' needs to send to site S two new vector triplets, one for F_j^{new} and one for F_k. It is not hard to see that the total (and parallel) computation cost for these operations is $O(|q||F_j|)$, while the communication cost is $O(|q|\mathsf{card}(\mathcal{F}_j))$. The analysis for mergeFragments(v) is similar and results are within the same bounds, where F_j now denotes the fragment that is the result of merging.

6. EXPERIMENTAL STUDY

We have implemented the algorithms of Sections 3 and 4. Below we present four of the conducted experiments. For our experiments we used 10 Linux machines (sites), distributed over a local LAN. We generated multiple XMark *"sites"* and in each experiment we assigned (fragments of) XMark *"sites"* to Linux machines. The reported times are averaged over multiple runs of each experiment.

Experiment 1: The objective of this experiment is twofold. First, we illustrate the effectiveness of Algorithm ParBoX when compared with the NaiveCentralized algorithm. Second, we study the effects of the query size (*i.e.,* its number of sub-queries $|\mathsf{QList}(q)|$) on query evaluation time. The fragment trees used in this experiment are similar to tree

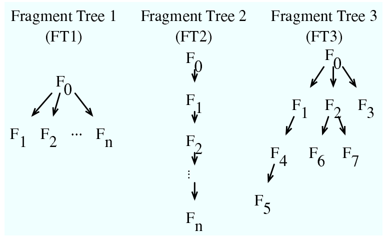


Figure 6: Samples of fragment trees used

FT1, shown in Fig. 6. Any fragment tree topology, with the same number of fragments could have been used here without affecting our results. This is because both ParBoX and NaiveCentralized communicate directly with all the sites holding fragments, irrespectively of the structure of the tree and the location of the fragments in the tree.

The experiment begins with a single fragment F_0 and in each iteration, it increases the number of fragments in the tree by one, by inserting a new fragment as a sub-fragment of F_0 and assigning it to a different machine. Each fragment F_j in the tree corresponds to a single XMark *site* and all the fragments have the same size *within* each iteration. The cumulative size of all the fragments is kept constant *across* iterations and equal to 50MB. So, in iteration 1, we have only fragment F_0 of size 50MB, while in iteration 2, we have fragments F_0 and F_1, each of size 25MB. In the last iteration, there are 10 fragments, each of size 5MB.

Figure 7 compares the evaluation times of ParBoX and NaiveCentralized of a query with $|\mathsf{QList}(q)| = 8$, for each iteration. The query is posed at the (coordinating) site holding fragment F_0. The figure clearly demonstrates both the benefits of using ParBoX, and that data transmission is prohibitively expensive. Notice that as the number of machines increases in each iteration, so does parallelism, resulting in reduced evaluation times. Parallelism proves especially beneficial in the first 4 iterations, but its gains diminish as increasingly smaller fragments are assigned to machines (iterations 8 - 10). Indeed, the evaluation time in iteration 8 (fragment size 6.3MB) is not significantly different from the one in iteration 10 (fragment size 5MB). In terms of the NaiveCentralized and NaiveDistributed algorithms, note that the query evaluation time of fragment F_0 (6.8 seconds) is a lower bound for both of these algorithms. The additional cost of NaiveCentralized in each iteration is the cost of sending the data to the coordinator. This cost is high for the first few iterations and starts to flatten out after iteration 6. Indeed, in iteration 2 we sent fragment F_1 to the coordinator, that is 25MB, while in iteration 3 we sent fragments F_1 and F_2, that is 34MB (9MB more). However, notice that in iteration 8, we sent fragments F_1 to F_7, that is 42MB, while in iteration 10 we sent fragments F_1 to F_9, that is 45MB (only 3MB more).

Figure 8 shows the evaluation times of ParBoX for queries whose ($|\mathsf{QList}(q)|$) sizes are 2, 8, 15 and 23. As expected, query evaluation time increases linearly with the size of the query and the benefits of parallelism are consistent throughout the different query sizes.

Experiment 2: The aim of this experiment is to highlight the differences between ParBoX and its variations. To this end, we use fragment trees similar to tree FT2, shown in

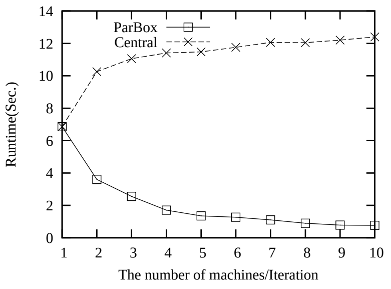


Figure 7: ParBoX **vs.** NaiveCentralized

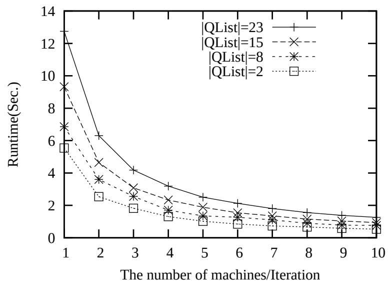


Figure 8: Scalability in query size

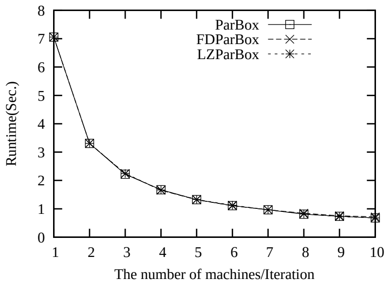


Figure 9: Query q_{F_0}

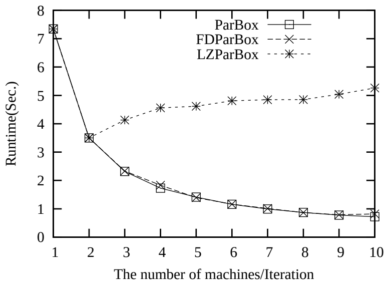


Figure 10: Query q_{F_n}

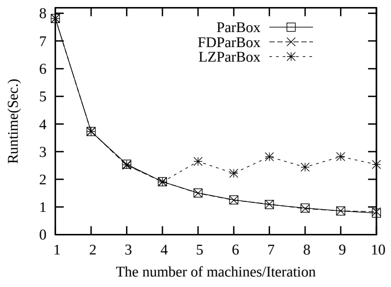


Figure 11: Query $q_{F_{\lceil n/2 \rceil}}$

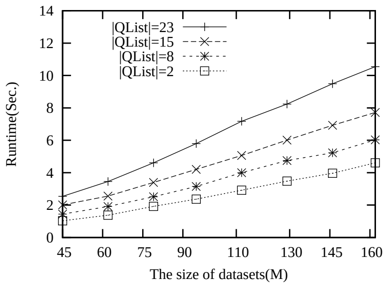


Figure 12: Scalability in data

Fig. 6. Like Experiment 1, we start with fragment F_0 and in each iteration j we add a new fragment F_j as a sub-fragment of fragment F_{j-1}. Note that fragment trees like FT2 appear often in practice. For example, in a temporal database each fragment can represent an XMark *"site"* at a point in time. Then, FT2 represents the version history of this XMark *"site"*. Like Experiment 1, we distribute 50MB of data evenly among the fragments in each iteration.

First consider a boolean query q_{F_0} that is executed in the (coordinating) machine holding fragment F_0 and it is satisfied by fragment F_0. Figure 9 shows the evaluation times of ParBoX, FullDistParBoX and LazyParBoX of q_{F_0}, for each iteration. Note in the figure that the evaluation (parallel computation) times for all algorithms are almost identical, while the total computation for the first two algorithms is much larger (not shown). This is because algorithms ParBoX and FullDistParBoX evaluate q_{F_0} over all the fragments of FT2, in parallel, while LazyParBoX by design is only evaluated in fragments F_0 and F_1. Recall that LazyParBoX initially evaluates a query only in the coordinator and in the fragments of depth 1 in the fragment tree. Since F_0 satisfies q_{F_0} no other fragment needs to be evaluated in LazyParBoX. The additional fragments considered in the first two algorithms have no overhead in the perceived evaluation time since (a) all fragments have the same size (b) each fragment is in a different machine (c) all evaluation is done in parallel. A few important things to note: First, in LazyParBoX only 2 machines evaluate q_{F_0} while all the other machines are idle. Second, network traffic and communication delays in our *partial evaluation algorithms* are negligible not only in this but in all the other experiments we conducted. This is because we *never* sent any data fragments between machines. In spite of the overall small traffic in our experiments, FullDistParBoX still results in at most half the traffic of ParBoX. These savings are due to FullDistParBoX not sending any variables.

Next consider a boolean query q_{F_n} that is executed in the (coordinating) machine holding fragment F_0 and is carefully selected so that it is satisfied by the last fragment F_n in each iteration. Figure 10 shows the evaluation times of ParBoX, FullDistParBoX and LazyParBoX of q_{F_n}, for each iteration. Note that in the first two iterations, by design, all algorithms evaluate q_{F_n}, in parallel, in both fragments F_0 and F_1. In subsequent iterations, algorithms ParBoX and FullDistParBoX both continue to evaluate q_{F_n}, in parallel, in all the fragments in FT2 and thus they have almost identical evaluation times. However, the evaluation time of LazyParBoX starts to increase since the algorithm has to consider *sequentially* the fragments in increasing depths of the tree, until it reaches fragment F_n where the query is satisfied. Due to this sequential access, the evaluation time of LazyParBoX is the sum of the evaluation times in fragments F_2 to F_n plus the (parallel) evaluation time of fragments F_0 and F_1. The increase between iterations is not linear since as the 50MB data are re-distributed between iterations, the total *additional* data that we need to sequentially evaluate between iterations i and $(i+1)$ are only $\frac{50}{i \times (i+1)}$. So, between iterations 2 and 3, the evaluation time is that of iteration 2 plus the cost of evaluating the query additionally over 8.3MB of data, while between iteration 9 and 10 the additional data are only 0.5MB.

Finally, consider a boolean query $q_{F_{\lceil n/2 \rceil}}$ that is executed in the (coordinating) machine holding fragment F_0 and in each iteration, it is satisfied by the $F_{\lceil n/2 \rceil}$ fragment, in the middle of ST2. Figure 11 shows the evaluation times of ParBoX, FullDistParBoX and LazyParBoX of $q_{F_{\lceil n/2 \rceil}}$, for each

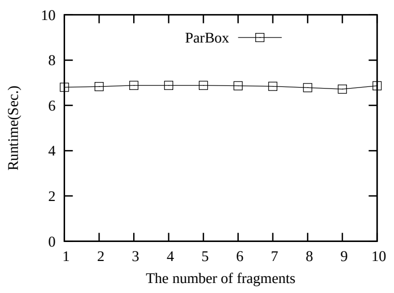


Figure 13: Varying the number of fragments

iteration. Again for the first two iterations, all three algorithms behave the same. Starting at iteration 3, the evaluation time of LazyParBoX starts to oscillate until it converges to a value of approximately 2.5 seconds. This is because starting at iteration 3, and every other iteration, we increase the depth of fragment $F_{\lceil n/2 \rceil}$ by one. So, the depth of $F_{\lceil n/2 \rceil}$ is 2 in iterations 3 and 4, while it is 3 in iterations 5 and 6 etc. If the depth of $F_{\lceil n/2 \rceil}$ is constant between two consecutive iterations i and $(i{+}1)$, the evaluation time of LazyParBoX improves since fragments up to the same depth of tree are considered in both iterations, but in iteration $(i + 1)$ less data are traversed (due to the re-distribution of data). This is the case for iteration pairs 3 and 4, 5 and 6 etc. Now, if the depth of $F_{\lceil n/2 \rceil}$ increases between two consecutive iterations i and $(i + 1)$, LazyParBoX considers one additional fragment in iteration $(i + 1)$. Therefore, there is a slight increase in evaluation time. This is the case for iteration pairs 4 and 5, 6 and 7 etc. As the size of fragments reduces in later iterations, the gains (losses) in evaluation times are also reduced. This is due to corresponding reduction in the size of less (additional) data considered between consecutive iterations. If we assume that a query is satisfied, on average, by a fragment close to the middle of the fragment tree, then this experiment shows that in LazyParBoX query evaluation is approximately 3 times slower than ParBoX. However, LazyParBoX saves half of the total computation done by ParBoX. Clearly, one is often willing to trade evaluation time for reduced site load.

Experiment 3: In the first two experiments the data size was kept constant. We now investigate the effect of data size on query evaluation times. Additionally, we use a more *natural* fragment tree like FT3, shown in Fig. 6. In each iteration, we increase the amount of data stored in the fragments of FT3. The fragments do not hold the same amount of data. For example, fragment F_0 is always around 10MB; fragment F_1 is the largest in the tree and its size ranges from 10MB up to 50MB, in 5MB increments per iteration; fragment F_2 ranges from 3.5MB to 15MB in 1.5MB increments; and fragment F_7 ranges from 700K to 3.7MB. The cumulative size of the fragments, in each iteration, is indicated in Fig. 12 and ranges from 45MB to 160MB. The figure shows the evaluation times of ParBoX for queries whose ($|\mathsf{QList}(q)|$) sizes are 2, 8, 15 and 23. As expected, for each query the evaluation time is linear *w.r.t.* the size of data. Moreover, as the query size increases, evaluation time increases gracefully over similarly sized data.

Experiment 4: In the previous experiments, each site was assigned one fragment. Here, we vary the number of fragments assigned to a site. Our objective is to show that the evaluation time of ParBoX depends on the cumulative size of the fragments assigned to a site and not on the number of these fragments. Indeed, this is the case as shown in Fig. 13. The figure shows the (almost constant) evaluation time in ParBoX of a query with $|\mathsf{QList}(q)| = 8$ for a single site, where the cumulative data assigned to it is 50MB. In iteration i the 50MB are split into i equi-sized fragments.

7. RELATED WORK

Partial evaluation has found applications ranging from compiler optimization to parallel evaluation of functional programming languages. Its relevance to query evaluation has surfaced from time to time, most notably in the closely related areas of query rewriting with views and deductive databases [9]. From a traditional functional programming perspective, our work is most closely related to the uses of partial evaluation in dataflow architectures [17] in which one evaluates some or all of the arguments of a function in parallel. The difference with our work is that traditional functional programming neglects the benefits of partial evaluation for functions accessing large data sets.

Distributed query evaluation has been extensively studied (see, *e.g.,* [19]). The key issue is minimizing data movement across sites as the communication costs are dominant [19]. Our algorithms minimize data movement by shipping partially evaluated functions (Boolean expressions) rather than data. Partial evaluation of XPath queries can also be combined with other distributed query processing techniques *e.g.,* hybrid shipping, two-phase optimization [19] and replication [1]. Also related is parallel query evaluation [7, 14], in particular in shared-nothing database machines. The main difference is the amount of communication involved. By shipping Boolean expressions rather than repartitioning/distributing data, we communicate relatively small amounts of data, and guarantee that each site is visited a *bounded* number of times. To our knowledge, no previous work has explored partial evaluation of XML queries in distributed or parallel database systems, by performing XML *tree traversal*, the most time consuming part, *in parallel.* The benefits of the partial evaluation technique are also evident in querying distributed catalogs (LDAP [3, 23]), and in XML query processing in systems where transmission delays are significant, such as in P2P systems. Our technique can be readily applied to evaluating Boolean XPath queries in a P2P system and it can be combined with existing techniques in this field (*e.g.,* [6, 8]).

XPath evaluation has been extensively studied for centralized systems, but the optimal strategies (*e.g.,* [10, 18]) may not work well in a distributed setting. Optimization techniques (*e.g.,* [5, 21]) are complementary to ours.

Most closely related to our work is [24], which studies distributed query evaluation on semistructured data with performance bounds on total network traffic and total computation. Its main technique is query decomposition: rewrite the input query Q into sub-queries based on the accessibility information of the distributed data, evaluate sub-queries on the corresponding data components to get partial results, and finally assemble the partial results to get the result of Q. The obvious difference is that in our work no query decomposition is necessary. Furthermore, [24] does not leverage (residual) *functions.* Another difference concerns the performance metric: [24] emphasizes the number of communication steps and counts a "broadcast" to (resp. a "gather" from) n sites as a single step. Though this is admittedly an approximation of the communication costs, it can be further improved. It also differs from our work in the queries and the distributed systems considered. Another related work is [20], which proposes the mutant query plans (MQP) for distributed XML queries. When a site receives a MQP it partially evaluates as much of the plan as it can, producing a new MQP plan and passes it to some other site that can continue processing. The sites evaluate queries in sequence rather than in parallel, as is the case in our work.

Incremental view maintenance has been studied for traditional databases (see [12]), data warehouses (*e.g.,* [26]), semistructured data (*e.g.,* [24]), and recently for XPath views of XML data [22]. These differ from our incremental algorithm in that they focus on centralized algorithms and/or different classes of views. An important criterion for incremental view maintenance in a de-centralized system is minimizing unnecessary visits to data fragments and remote sites, which our algorithm provides. To our knowledge, our algorithm is the first for incremental maintenance of Boolean XPath views in a de-centralized system.

8. CONCLUSIONS

We have shown that partial evaluation is effective in processing XML queries in a de-centralized system. We have proposed the first evaluation and incremental maintenance algorithms with *performance guarantees* for Boolean XPath queries. Our experimental study has verified the effectiveness of our technique. In addition to its application to query processing in such systems, we believe that the technique has the potential to be useful in other applications, in distributed and centralized systems alike.

We plan to extend the current work in a number of directions. First, we plan to extend our algorithms to handle more general queries in XPath and XQuery. A recent extension is capable of processing data selection XPath queries with the performance guarantee that each site is visited at most twice, and thus demonstrates the potential of the technique to process non-Boolean queries. Second, we aim to incorporate other optimization techniques for XML query processing, in the presence of replication. Third, we plan to apply the technique to processing queries over large documents in (centralized) native XML stores. Finally, it is not difficult to identify other situations in which partial evaluation could yield major benefits in a de-centralized system. For example, numerical and aggregating computations over large data sets can benefit from the technique. We believe that the general topic deserves a broader study.

9. REFERENCES

[1] Serge Abiteboul, Angela Bonifati, Gregory Cobena, Ioana Manolescu, and Tova Milo. Dynamic XML documents with distribution and replication. In *SIGMOD*, 2003.

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

[3] Sihem Amer-Yahia, Divesh Srivastava, and Dan Suciu. Distributed evaluation of network directory queries. *TKDE*, 16(4):474–486, 2004.

[4] Jan-Marco Bremer and Michael Gertz. On distributing XML repositories. In *WebDB*, pages 73–78, 2003.

[5] E. Colen, H. Kaplan, and T. Milo. Labeling dynamic XML tree. In *PODS*, 2002.

[6] Adina Crainiceanu, Prakash Linga, Johannes Gehrke, and Jayavel Shanmugasundaram. Querying peer-to-peer networks using P-Trees. In *WebDB*, 2004.

[7] David J. DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6), 1992.

[8] Prasanna Ganesan, Mayank Bawa, and Hector Garcia-Molina. Online balancing of range-partitioned data with applications to Peer-to-Peer systems. In *VLDB*, 2004.

[9] Parke Godfrey and Jarek Gryz. A strategy for partial evaluation of views. In *Intelligent Information Systems*, 2000.

[10] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing XPath queries. In *VLDB*, 2002.

[11] Jim Gray. Where the rubber meets the sky: The semantic gap between data producers and data consumers. In *SSDBM*, page 3, 2004.

[12] A. Gupta and I. Mumick. *Materialized Views*. MIT Press, 2000.

[13] Alon Y Halevy. Theory of answering queries using views. *SIGMOD Record*, 29(4), 2001.

[14] H. Hsiao and D. J. DeWitt. Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines. In *ICDE*, 1990.

[15] G. Huck, I. Macherius, and P. Fankhauser. PDOM: Lightweight persistency support for the document object model. In *OOPSLA*, 1999.

[16] H. V. Jagadish, Laks V. S. Lakshmanan, Tova Milo, Divesh Srivastava, and Dimitra Vista. Querying network directories. In *SIGMOD*, pages 133–144, 1999.

[17] Neil D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3), 1996.

[18] Christoph Koch. Efficient processing of expressive node-selecting queries on XML data in secondary storage: A tree automata-based approach. In *VLDB*, 2003.

[19] D. Kossman. The State of the Art in Distributed Query Processing. *ACM Computing Surveys*, 32(4):422–469, 2000.

[20] Vassilis Papadimos and David Maier. Distributed queries without distributed state. In *WebDB*, pages 95–100, 2002.

[21] Prakash Ramanan. Efficient algorithms for minimizing tree pattern queries. In *SIGMOD*, 2002.

[22] Arsany Sawires, J Tatemura, Oliver Po, Divyakant Agrawal, and K. S. Candan. Incremental maintenance of path expression views. In *SIGMOD*, 2005.

[23] Mark Smith and Timothy A. Howes. *LDAP : Programming Directory-Enabled Apps*. Sams, 1997.

[24] Dan Suciu. Distributed query evaluation on semistructured data. *TODS*, 27(1):1–62, 2002.

[25] Wanhong Xu and Z. Meral Özsoyoglu. Rewriting XPath queries using materialized views. In *VLDB*, 2005.

[26] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View maintenance in a warehousing environment. In *SIGMOD*, 1995.