Distributed query evaluation with performance guarantees

Distributed Query Evaluation with Performance Guarantees

Gao Cong
Microsoft Research Asia
gaocong@microsoft.com

Wenfei Fan
University of Edinburgh &
Bell Laboratories
wenfei@inf.ed.ac.uk

Anastasios Kementsietsidis
University of Edinburgh
akements@inf.ed.ac.uk

Abstract

Partial evaluation has recently proven an effective technique for evaluating *Boolean* XPath queries over a fragmented tree that is distributed over a number of sites. What left open is whether or not the technique is applicable to generic data-selecting XPath queries. In contrast to Boolean queries that return a single truth value, a generic XPath query returns a set of elements, and its evaluation introduces difficulties to avoiding excessive data shipping. This paper settles this question in positive by providing evaluation algorithms and optimizations for generic XPath queries in the same distributed and fragmented setting. These algorithms explore *parallelism* and retain the performance guarantees of their counterpart for Boolean queries, *regardless of* how the tree is fragmented and distributed. First, each site is visited at most three times, and down to at most twice when optimizations are in place. Second, the network traffic is determined by the final answer of the query, rather than the size of the tree, without incurring unnecessary data shipping. Third, the total computation is comparable to that of centralized algorithms on the tree stored in a single site. We show both analytically and experimentally that our algorithms and optimizations are scalable and efficient on large trees and complex XPath queries.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—Query Processing

General Terms

Algorithm, Performance

Keywords

Distributed XML documents, Xpath Queries, Parallel Query Processing

1. Introduction

Partial evaluation has recently found an effective application in the evaluation of *Boolean* XPath queries over an XML tree that is fragmented, both horizontally and vertically, and is distributed over a number of sites [5]. The

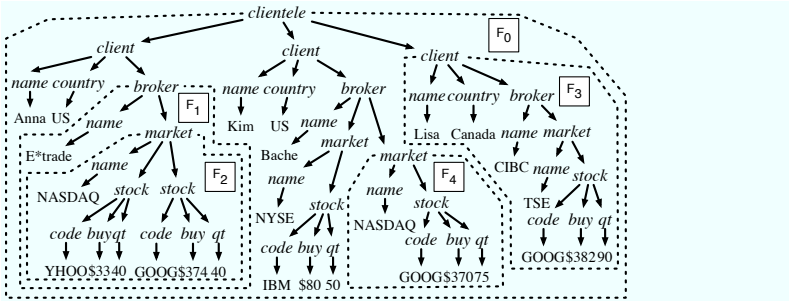


Figure 1: Investment company clientele

rough idea of partial evaluation (see [18] for a survey) is the following. Consider a function $f(\bar{x}_1, \bar{x}_2)$. Assumes that we are given part of its input, say only argument $\bar{x}_1$. Then, partial evaluation specializes function f with respect to the known argument $\bar{x}_1$, *without* waiting for the other argument $\bar{x}_2$. That is, it performs the part of f's computation that depends only on $\bar{x}_1$, and generates a partial answer, *i.e.,* a *residual function* f' that depends on the as yet unavailable argument $\bar{x}_2$. To illustrate its application in Boolean XPath evaluation, let us consider an example.

Consider the XML tree T shown in Fig. 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 decomposed into a number of sub-trees, or *fragments*, and are distributed over the Internet for geographical or administrative reasons [3], a setting commonly found in e-commerce, Web services, or while managing large-scale network directories [16]. In Fig. 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 Fig. 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 that is posed at site S_0. When being evaluated at the root of the tree T, the query returns a

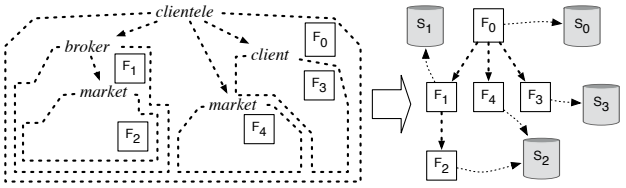


Figure 2: Tree fragmentation and fragment tree

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 [20]. This would require a single depth-first traversal of T, visiting each node exactly once, something we cannot expect to improve upon. However, this seemingly "optimal" approach would visit fragments $F_0, F_1, F_2, F_1, F_0, F_4, F_0, F_3, F_0$ in that order, visiting F_0 four times and F_1 twice.

An algorithm, called ParBoX, was developed in [5] based on partial evaluation, which evaluates Boolean XML queries over a fragmented tree that is distributed over a number of different sites. The algorithm partially evaluates the whole query Q, in parallel, over each fragment of the tree. Since each fragment contains only part of the tree, this partial evaluation of Q over 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 single coordinator site and are *composed* resulting in the final answer to Q. The ParBoX algorithm has a number of desirable properties, namely: (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 fragmented, what the sizes of these fragments are, or how they are assigned to sites.

A question left *open* in [5] is how and under which circumstances the partial evaluation technique can be used to evaluate generic *data selecting* XPath queries. The need for the extension is evident since most XPath queries in practice are data-selecting queries, which find a *set of* XML elements rather than return a single truth value. This highlights the demand for an efficient algorithm to evaluate generic XML queries in distributed settings. Of course, such an extension should not come at a high price. It would be desirable if, while supporting generic XPath queries, we could retain properties comparable to those of the ParBoX algorithm.

It is, however, nontrivial to extend the technique to data-selecting queries. Consider a data selecting XPath query Q' = *//broker*[*//stock/code/text()* ="GOOG"]/*name*. A direct extension of the technique of [5] would send Q' to every site and partially evaluate Q'. However, what should be returned by each site as a partial answer, namely, residual function? Note that the final answer of Q' is a set of elements, and thus using the technique of [5], each site may send a set of elements that are *probably* in the answer of Q', and let the coordinator do the checking. This leads to excessive data shipping and in the worst case may end up shipping the entire tree to the coordinator. 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 [20], instead of a single pass as for Boolean XPath queries.

To this end we develop algorithms and optimizations for evaluating generic data selecting XPath queries in the same fragmented and distributed setting of [5]. Based on partial evaluation, our algorithms retain the properties of [5]. Our contributions can be summarized as follows:

• We introduce the *first* partial evaluation algorithm for generic XPath queries in the distributed setting. We support a fragment of XPath with the downward axes, which covers most XPath queries used in practice [15] and includes the Boolean XPath of [5] as a special case. Our first evaluation algorithm *guarantees* that each site is visited at most *three times* and our optimized algorithm 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 computation 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.

• Our second contribution is an optimization technique on the performance of our basic evaluation algorithm which reduces the communication and computation costs.

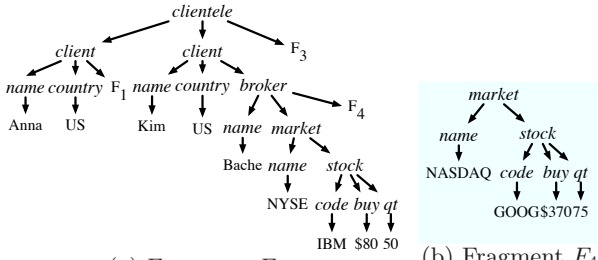
• Our third contribution is a detailed experimental study, using our implemented partial-evaluation XPath query engine. Our experiments demonstrate the full potential of partially evaluating XPath queries on distributed stores.

We believe that our proposed techniques are promising for efficiently evaluating generic XPath queries in distributed systems with *performance guarantees*. We expect that the evaluation algorithms also shed light on evaluating XPath queries on large-scale datasets in centralized systems. Indeed, when the whole tree does not fit in main memory, through fragmentation we are able to load each time from secondary storage a different fragment of the tree into main memory. Our partial evaluation techniques help reduce at least the cost of swapping the fragments.

Organization. Section 2 discusses XML tree fragmentation and reviews the class of XPath queries considered in this paper. Section 3 presents a basic evaluation algorithm for XPath queries based on partial evaluation, referred to as PaX3, which directly employs ParBoX to evaluate XPath qualifiers and requires at most *three* visits at each site. We start with this basic algorithm *for its conceptual clarity and to simplify the discussion*. In Section 4 we then present an improved version of PaX3, which visits each site at most *twice*. An optimization technique is given in Section 5. An experimental study is provided in Section 6, followed by related work in Section 7 and conclusions in Section 8.

2. Preliminaries

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



(a) Fragment F_0 (b) Fragment F_4

Figure 3: Example fragments

2.1 XML Tree Fragmentation

We consider settings in which an XML tree T is decomposed into a set $\mathcal{F}_T$ of disjoint trees, or *fragments*. Each fragment $F_i \in \mathcal{F}_T$ can be stored in a different site. We *do not impose any constraints* on the fragmentation: we allow for an 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: this is determined by the system. Hence our fragmentation setting is the most generic possible. As an example, the tree in Fig. 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 fragmentation of a tree T induces another tree, called a *fragment tree*, which represents the relationship between the different fragments of T. We are going to use $\mathcal{F}_T$ to denote both the set of fragments of a tree T and the corresponding induced fragment tree representation. It will be clear from the context which of the two notions we refer to. The fragment tree $\mathcal{F}_T$ for tree T of Fig. 1 is shown to the right of Fig. 2. We call *root* fragment, the fragment at the root of the fragment tree which also contains the root of tree T. In Fig. 2, this is fragment F_0. Furthermore, 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. 1, fragment F_2 is a sub-fragment of F_1 which, in turn, is a sub-fragment 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 Fig 2. For example, fragment F_3 is stored in site S_3 while both fragments F_2 and F_4 are stored in site S_2. Given the distribution of fragments, we need to maintain the relationship between a fragment and its sub-fragments so as to preserve the structure of the original tree T. To this end, given a fragment F_j and its sub-fragment 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 Fig. 3(a) fragment F_0 has virtual nodes representing fragments F_1, F_3 and F_4. While traversing fragment F_0 stored 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 Fig. 3(b). We refer to a fragment that has no sub-fragments as a *leaf* fragment. In Fig. 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:

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

where Q is a *path expression* defined in terms of the empty path ϵ (*self*), label A (tag), wildcard $*$, the *descendant-or-self-axis* '//', child '/', 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 *name* of a *broker* through which GOOG stocks are purchased, but no YHOO stocks, we write a 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, *i.e.,* when T is not decomposed and distributed, $\mathsf{val}(Q, r)$ can be computed in $O(|T|\ |Q|)$ time [11], where r is the root of T.

The class $\mathcal{X}$ subsumes twig queries [4] and Boolean XPath studied in [5]. Note that although our query language supports only the self, child and descendant XPath axes, this is usually sufficient since the majority of XPath queries use the downward axes [15].

Similar to [5], we normalize each query Q in $\mathcal{X}$ and convert it into a normal form (although our normal form is slightly different here). Specifically, 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:

$\mathsf{normalize}(\epsilon) = \epsilon$; 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. In the sequel, we consider $\mathcal{X}$ queries in the normal form.

Striking out all the qualifiers in $\beta_1/\ldots/\beta_n$ we get a "path" $\eta_1/\ldots/\eta_n$, where η_i is either A, ϵ, $*$, or //. We refer to $\eta_1/\ldots/\eta_n$ as the *selection path* of query Q. For example, the selection path of query Q_1 given above 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 sub-queries of the qualifiers of Q. We sort $\mathsf{QVect}(Q)$ in a topological order such that for any sub-queries q_1, q_2, if q_1 is a sub-query 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:

$\mathsf{normalize}(Q) = \mathit{client}/\epsilon[\mathit{country}/\epsilon[\mathit{text}()=\text{“US”}]]/\mathit{broker}/$
$\epsilon[\mathit{market}/\mathit{name}/\epsilon[\mathit{text}() = \text{“NASDAQ”}]]/\mathit{name}$

We decouple the selection path */client/broker/name* of the query from the qualifiers $[*/\epsilon[country/\epsilon[text()=\text{“US”}]]$ and $[*/\epsilon[market/name/\epsilon[text() = \text{“NASDAQ”}]]$. Then, vectors $\mathsf{SVect}(Q)$ and $\mathsf{QVect}(Q)$ are as follows:

$\mathsf{SVect}(Q) = [q_1, q_2, q_3]$ where

$q_1 = \mathit{client}, \quad q_2 = q_1/\mathit{broker}, \quad q_3 = q_2/\mathit{name}$

$\mathsf{QVect}(Q) = [q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9]$, where

$q_1 = \mathit{country}$, $q_2 = [\mathit{text}()=\text{“US”}]$, $q_3 = q_1/\epsilon[q_2]$, $q_4 = */\epsilon[q_3]$,
$q_5 = \mathit{name}$, $q_6 = [\mathit{text}()=\text{“NASDAQ”}]$, $q_7 = q_5/\epsilon[q_6]$,
$q_8 = \mathit{market}/q_7$, $q_9 = */\epsilon[q_8]$

where the first four entries in $\mathsf{QVect}(Q)$ are for the first qualifier, while the remaining five are for the second. □

3. Query Evaluation

Consider an $\mathcal{X}$ query Q submitted to a site S_Q and the query is to be evaluated over a fragmented and distributed XML tree T. A naïve evaluation of Q requires collecting to site S_Q all the fragments of tree T, identified by the fragment tree $\mathcal{F}_T$. Once collected, tree T is reconstructed by its fragments and a *centralized* algorithm is used to evaluate Q over T. We refer to this approach as NaiveCentralized. The authors of [5] have already shown experimentally that this approach is inefficient, even for the case of simple Boolean queries. Although there are a number of efficient XPath evaluation engines for centralized XML stores, in research (e.g. the algorithm of [11]) and beyond (e.g. Saxon [24], Xalan [29]), the efficiency of these evaluators becomes apparent only after site S_Q gets all the data. This incurs large overhead in network traffic, since fragments are transmitted over the network each time a query is executed. Furthermore, the NaiveCentralized algorithm assumes that site S_Q has main memory big enough to fit T. Worse still, as remarked earlier, security or privacy reasons may prevent entire fragments to be shipped from one site to another.

To cope with this we propose the Parallel XPath (PaX3) evaluation algorithm, based on *partial evaluation*. The PaX3 Algorithm *guarantees* the following:

1. Each site is visited only at most three times, 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* and *the size of the query answer* rather than the XML tree.

Thus algorithm PaX3 maintains all the desirable properties of the ParBoX algorithm, while it allows evaluating generic data-selecting XPath queries, instead of just Boolean ones.

We start by presenting the PaX3 algorithm to focus on the main idea of the partial evaluation technique. In the next section we will present a refined version of PaX3, called PaX2, which guarantees that each site is visited at most *twice* while retaining the other properties of PaX3.

The PaX3 algorithm is initiated at site S_Q where the query Q is issued. Without loss of generality, we assume S_Q to be the site storing the root fragment of the tree T. As shown in Fig. 4, the algorithm consists of 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 XML tree T. Specifically:

Stage 1: The objective of this stage is to (partially) evaluate the *qualifiers* of query Q. For each node of each fragment, we partially evaluate qualifiers by using our own extension of the ParBoX algorithm [5]. Intuitively, 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, *i.e.,* Boolean formula, will be fully known for all nodes by the beginning of the next stage.

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 to be part of the answer. Again, candidacy depends on the value of a Boolean formula.

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

Note that the ParBoX algorithm of [5] 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.

In what follows, we describe each stage in more detail.

3.1 Qualifier Evaluation

During Stage 1 of Algorithm PaX3, the evaluation of qualifiers is performed through our extension of the ParBoX algorithm [5], which we present here briefly since the extended ParBoX is only one part of our evaluation algorithm and it is not a main contribution of our work. Our extensions include a more expressive language in the qualifiers, which includes arithmetic comparisons, and the ability to evaluate multiple top-level qualifiers in a query while in contrast, ParBoX only needs to compute a single top-level qualifier. Obviously, if a query has no qualifiers, the whole stage can be skipped and the evaluation proceeds with Stage 2.

Consider a simple query Q over T. For simplicity, assume that Q has a single qualifier. An efficient evaluation of $\mathsf{QVect}(Q)$ for all the nodes v of T requires a single bottom-up traversal of T. During the traversal, at a node v we compute the values of all the sub-queries in $\mathsf{QVect}(Q)$ and store them in a vector QV_v associated with v. For this computation, we often need to consult the (already computed) values of the $\mathsf{QVect}(Q)$ sub-queries at the children and descendants of v. Only two additional vectors are required to store these values, namely, vectors QCV_v and QDV_v, respectively. Intuitively, for each sub-query q in $\mathsf{QVect}(Q)$, $QCV_v(q)$ 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. Now, if Q has a single qualifier, then the value of the qualifier at v is determined by the value of the last entry in QV_v. If Q has more than one qualifiers, as is the case in Example 2.1, then the truth value of each qualifier is determined by the entry in QV_v corresponding to

Procedure PaX3

Input: An XPath query Q and a fragmented tree T
Output: The answer (set of nodes) ans of Q over T

```
    /* Stage 1*/
1.  for each site S_i in F_T do
2.      exec(S_i, evalQual, QVect(Q)) in parallel;
3.      for each fragment F_j stored in S_i do
4.          annotate F_T with (QV_Fj, QCV_Fj, QDV_Fj);
5.  evalFT(F_T);
    /* Stage 2*/
6.  for each fragment F_j stored in S_i do
7.      for each sub-fragment F_k of F_j do
8.          send (QV_Fk, QCV_Fk, QDV_Fk) to S_i;
9.  for each site S_i in F_T do
10.     exec(S_i, evalSelQ, SVect(Q)) in parallel;
11.     for each fragment F_j stored in S_i do
12.         for each sub-fragment F_k of F_j do
13.             annotate F_T with SV_Fk;
14. evalFT(F_T);
    /* Stage 3*/
15. for each subfragment F_k of a fragment F_j do
16.     send SV_Fk to site S_l storing F_k;
17. for each site S_i in F_T do exec(S_i, collectAns);
18. receive ans from each site S_i
```

(a) PaX3 algorithm executed at site S_Q

Procedure evalSelQ

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

```
1. for each fragment F_j assigned to S_i do
2.     cans := ∅;
3.     tempSet := topDown(root(F_j), SVect(Q));
4.     returnSet := returnSet ∪ tempSet;
5. send returnSet to site S_Q;
```

Procedure topDown

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

```
1.  initStack();
2.  for each query q_i in SVect(Q) from left to right do
3.    case q_i of
4.      t : SV_v(q_i) := term(v, t);
5.      q_j/t: SV_v(q_i) := evalFM(stack(SV(q_j)), term(v, t), ∧);
6.      q_j//: SV_v(q_i) := evalFM(stack(SV(q_i)), SV_v(q_j), ∨);
7.    SV_v(q_i) := evalFM(SV_v(q_i), assocQual(QV_v), ∧);
8.  pushStack(v, SV_v)
9.  if SV_v(|SVect(Q)|) = true then ans := ans ∪ v;
10. elseif SV_v(|SVect(Q)|) is a formula then
11.       cans := cans ∪ (v, SV_v(|SVect(Q)|));
12. if v is a virtual node then returnSet := returnSet ∪ SV_v
13. for each child w of v do
14.     SV_w := topDown(w, SVect(Q));
15. 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.      ε: return true;
3.      *: return true;
4.      label() = l: return compareString(label(), l);
5.      text() = str: 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(|SVect(Q)|)) in cans do
2.     if unify(SV_v(|SVect(Q)|)) = true then ans := ans ∪ v;
3. return ans
```

(b) PaX3 algorithm executed at participating site

Figure 4: The PaX3 Algorithm

the sub-query which represents that qualifier. In our example, the value of the first qualifier is determined by q_4 while that of the second is determined by entry q_9.

The above simple procedure assumes that tree T is not fragmented. In the presence of fragmentation, we need to perform a bottom-up evaluation of QVect(Q) for each fragment. An immediate problem then is how to compute, during the traversal of the fragment, the value of QVect(Q) given that parts of the tree are *missing* and are replaced by virtual nodes. Since each fragment is processed in parallel, the values of QVect(Q) are unknown for the virtual nodes and their corresponding fragments and, under normal circumstances, until we learn these values from the site holding the fragment for the virtual node, we cannot proceed with the evaluation. *This* is the point where partial evaluation comes into play. The key idea in the partial evaluation of qualifiers is to introduce *Boolean variables*, one for each missing value of QVect(Q) at each virtual node.

Example 3.1: Consider query Q from Example 2.1. Then, the following vectors result in by evaluating QVect(Q) over the nodes of the leftmost client of fragment F_0 in Fig. 3(a):

$QV_{name} =$ <0, 0, 0, 0, 1, 0, 0, 0, 0>
$QV_{country} =$ <1, 0, 1, 0, 0, 0, 0, 0, 0>
$QV_{F_1} =$ <x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9>
$CQV_{F_1} =$ <cx_1,cx_2,cx_3,cx_4,cx_5,cx_6,cx_7,cx_8,cx_9>
$QV_{client} =$ <0, 0, 0, 1, 0, 0, 0, 0, x_8>

Recall from Example 2.1 that $q_4 = */\epsilon[q_3]$. That is, at a node v the value of the first qualifier (entry q_4) depends on the value of entry q_3 at the child nodes of v. Indeed, notice that the first qualifier (value of q_4) is true in QV_{client} since *client* has a child node *country* whose corresponding text is "US" (entry q_3 in $QV_{country}$). We also consider the value of the second qualifier (value of q_9) in QV_{client} although, in practice, this qualifier is *associated* with *broker* nodes in our query. From Example 2.1 we know that $q_9 = */\epsilon[q_8]$, that is, at a node v the value of the second qualifier (entry q_9) depends on the value of entry q_8 at the child nodes of v. In the specific *client* node, the value of this qualifier depends on variable x_8, that is, it depends on whether or not the virtual node F_1 represents such a fragment that its root node is *market* with a *name* child that has a value of "NASDAQ". Note that for virtual node F_1 we introduce fresh variables since we do not know the value for any of the entries in the vector. Note that there are dependencies between some of the introduced variables. For example, the value of the first qualifier in F_1, that is variable x_4, is equal to cx_3 since x_4 is true in node F_1 as long as node F_1 has a child node whose label is *country* and whose value is "US" (variable cx_3). □

In the algorithm (shown in Fig. 4), the bottom-up partial evaluation of qualifiers is initiated in site S_Q which makes a remote procedure call (Fig. 4(a), line 2) to all the sites holding at least one tree fragment. The actual evaluation, outlined in the previous paragraphs, is performed by Procedure evalQual (not shown due to space constraints) which returns the triplet of vectors $(QV_{F_j}, QCV_{F_j}, QDV_{F_j})$ corresponding to the root node of fragment F_j. Each site returns its triplet(s) to site S_Q, one triplet per fragment. Then in site S_Q, Procedure evalFT is executed (not shown due to space constraints). Remember that we introduce variables during the partial evaluation of fragments, for each virtual node of a fragment. During the computation of qualifiers, these variables are often composed and result in complex

Boolean formulas that appear as values in some vector entry [5]. The objective of Procedure evalFT is to use information from all the fragments and, by unifying variables, to compute the Boolean values for these vector entries. In more detail, Procedure evalFT considers the fragment tree which is now annotated with vector triplets. The procedure requires a single bottom-up traversal of the fragment tree to unify all variables and compute the Boolean formula truth values. Note that vectors of leaf fragments in the fragment tree contain no variables (since they do not have any virtual nodes), as shown in our example for fragments F_2, F_3 and F_4. During the bottom-up traversal of $\mathcal{F}_T$, Procedure evalFT uses the Boolean values of the leaf fragments to unify the variables of the vectors that belong to the parent fragments in $\mathcal{F}_T$. The procedure continues in this fashion until it reaches the root of $\mathcal{F}_T$.

Example 3.2: After Procedure evalQual concludes in fragments F_1 and F_2, the following are some of the vectors that are computed for nodes of the two fragments.

Vectors computed in fragment F_1
$QV_{broker} =$ $<0,\ 0,\ 0,\ \ y_3,\ 0,\ \ 0,\ 0,\ 0,\ \ y_8>$
$QV_{F_2} =$ $< y_1, y_2, y_3,\ y_4,\ \ y_5,\ y_6, y_7, y_8, y_9 >$

Vectors computed in fragment F_2
$QV_{market} =$ $<0,\ 0,\ 0,\ \ 0,\ \ 0,\ \ 0,\ 0,\ 1,\ 0>$
$QV_{name} =$ $<0,\ 0,\ 0,\ \ 0,\ \ 1,\ \ 0,\ 1,\ 0,\ 0>$

Vectors QV_{broker} and QV_{market} are computed for the roots of the two fragments, respectively. Vector QV_{F_2} corresponds to virtual node F_2 of fragment F_1 where a distinct variable is introduced for each unknown vector value. Vector QV_{name} is computed at the *name* node of fragment F_2. Note that q_8 is true in QV_{market}, i.e., the *market* node has a *name* child node with value "NASDAQ". The computation of q_8 relies, in turn, on the precomputed (by the bottom-up evaluation) value of entry q_7 from QV_{name}.

As described earlier, vectors QV_{broker} and QV_{market} are sent to site S_Q and are used as input to Procedure evalFT. The procedure unifies the variables in the received vectors by matching virtual nodes to root fragment nodes. That is, it determines that the vector which introduced the y_i variables in fragment F_1 (resp. vector QV_{F_2} for virtual node F_2) is essentially vector QV_{market} corresponding to the root node *market* of fragment F_2. By matching the two vectors, the procedure unifies variable y_8 to true (entry q_8 of QV_{market}). This, in turn, implies that entry q_9 in QV_{broker} is also true and therefore the second qualifier of the initial query is true for the corresponding *broker* node. □

3.2 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 (Fig 4(a), line 6), i.e., the truth values of vector triplets $(QV_{F_k}, QCV_{F_k}, QDV_{F_k})$ for each sub-fragment F_k of F_j (Fig 4(a), lines 6-8). The received vector triplets 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 (Fig 4(a), lines 9-10) to all the sites holding at least one tree fragment.

We now examine in more detail the partial evaluation of a selection path. Consider a query Q over T and, in particular, consider the selection path vector $\mathsf{SVect}(Q)$ of Q. An efficient evaluation of $\mathsf{SVect}(Q)$ over a tree T requires a single top-down (depth-first) traversal of T (Fig 4(b), Procedure topDown). For each node v encountered during the traversal and for each sub-query 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 topDown, lines 2-6). This computation often requires to consult the (already computed) values of the $\mathsf{SVect}(Q)$ sub-queries 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 each time the vector at the top of the stack *summarizes* the information for all vectors in the stack. More specifically, when v is being processed, SV_p is the top of the stack, where p is the parent of v. With this we compute $SV_v(q_i)$ 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 topDown and procedure term of Fig 4(b)). 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* term (v, t) are *true*. Here function evalFM (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 $\mathsf{SVect}(Q)$. We do this without imposing any additional overhead, in terms of space, to our vectors which are still linear in the size of the query Q. At the same time, we manage to save computation time since we do not have to go through the whole stack for each node under consideration. Also note that unlike the evaluation of qualifiers which require three vectors per node, here we only maintain a *single* vector per node, an improvement of our algorithm compared to ParBoX [5]. At end of the computation at node v, we consult the *last entry* in SV_v, denoted by $SV_v(|\mathsf{SVect}(Q)|)$. If the value of this entry is *true* then node v is part of the answer for query Q and is put in the set ans, otherwise it is not.

There are two more things to consider during the evaluation of selection paths. First, we need to consider qualifiers. Recall that qualifiers and selection paths are evaluated independently by our algorithm. Also, note that the truth values of all qualifiers, in all the nodes, are known after the end of Stage 1. We maintain the relationship between the selection $\mathsf{SVect}(Q)$ and qualifier part $\mathsf{QVect}(Q)$ of query Q through Procedure assocQual (not shown). The procedure returns for each entry of $\mathsf{SVect}(Q)$ the qualifier entry in $\mathsf{QVect}(Q)$ for which the truth value we need to check. To determine the truth value the qualifier associated to q_i at a node v and entry q_i of $\mathsf{SVect}(Q)$ (Procedure topDown, line 7), we only need to instantiate the variables in QV_v with the corresponding truth values in QV_{F_k} for sub-fragments F_k, which were received from site S_Q at the beginning of the second stage.

Example 3.3: For simplicity, ignore for a moment the fragmentation of the tree in Fig. 1 and assume that we evaluate $\mathsf{SVect}(Q)$ from Example 2.1 over the three *client*s of the tree. Then, the following are the $\mathsf{SVect}(Q)$ 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 *country* evaluate to true (from Stage 1). As a result, 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 thing we need to consider is fragmentation. At the beginning of the top-down traversal, given the root node r of a fragment F_j, we do not know the $\mathsf{SVect}(Q)$ 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 $\mathsf{SVect}(Q)$ 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 $\mathsf{SVect}(Q)$ vectors, namely, returnSet, one vector for each sub-fragment (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 3.4: After Procedure topDown concludes in fragment F_1, the following $\mathsf{SVect}(Q)$ vectors are computed for the nodes in F_1.

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

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 3.4, minimum network traffic while computation costs remain optimal.

The uncertainty of what precedes node *broker* is propagated in both $\mathsf{SVect}(Q)$ 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. □

3.3 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 (Fig 4(a), line 15), i.e., the truth values of vector SV_{F_k}. Note that although vector SV_{F_k} was sent to S_Q from the 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 determine which candidate answers in cans are real answers of Q (Fig. 4(b), 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 determines, in turn, that node *name* is an answer to query Q and thus it sends this node back to site S_Q.

3.4 Analysis

To compare with [5], for the analysis of the PaX3 algorithm, we consider the communication cost of the algorithm as well as its *total* and *parallel* computation costs. The total computation cost is the sum of the computation performed at all the sites that participate in the evaluation. 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 measures the perceived execution time of the algorithm and therefore it more accurately describes its performance.

Communication cost. As shown in [5], 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 the query Q. It is easy to see that the second stage of PaX3 also has communication cost $O(|Q|\ |\mathcal{F}_T|)$. Finally, the last stage has communication cost $O((|Q|\ |\mathcal{F}_T|)\ +\ |\mathsf{ans}|)$. Therefore, the total communication cost of PaX3 is $O((|Q|\ |\mathcal{F}_T|)\ +\ |\mathsf{ans}|)$.

Note that when we execute a query Q in a distributed environment with at most $|\mathcal{F}_T|$ 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, it is obvious that one cannot avoid the cost of retrieving the actual answers to our query (cost $O(|\mathsf{ans}|)$). In this sense, a communication cost $O((|Q|\ |\mathcal{F}_T|)\ +\ |\mathsf{ans}|)$ 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 $O(|Q|\ |T|)$. Note that this coincides with the cost of executing a query Q over a tree T in a central site [11]. Therefore, the distribution of computation does not incur much extra costs in terms of total computation.

Parallel computation cost. As more than one fragments 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.

Summary. With the exception of the necessary $O(|\mathsf{ans}|)$ cost incurred by transmitting query answers and the necessary at most two visits per site, the costs of PaX3 coincide with those of ParBoX. In short, we have proposed an algorithm to partially evaluate a larger, and more useful,

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_Fj, QCV_Fj, QDV_Fj);
5.      for each sub-fragment F_k of F_j do
6.          annotate F_T with SV_Fk;
7.  evalFT(F_T);
    /* State 2 */
8.  for each fragment F_j stored in S_i do
9.      for each sub-fragment F_k of F_j do
10.         send (QV_Fk, QCV_Fk, QDV_Fk) to S_i;
11.         send SV_Fk to site S_l storing F_k;
12. for each site S_i in F_T do exec(S_i, collectAns);
13. receive ans;
```

Figure 5: The PaX2 Algorithm

fragment of queries than those considered by ParBoX yet we provided *comparable* performance guarantees. Moreover, we guarantee *minimum* tree data transmission since the only tree data transmitted by PaX3 are the actual query answers.

4. Improved Algorithm

We next present Algorithm PaX2, which needs two stages and at most *two* visits of each site, one less than PaX3.

The main idea behind algorithm PaX2 is to *combine* the first two stages of algorithm PaX3, i.e., evaluation of qualifiers and that of selection paths, into a single stage. As shown in Fig. 5, the algorithm starts with letting querying site S_Q make a remote procedure call to all the sites holding fragments (lines 1-2). At each such site, Procedure evalXPath (not shown due to space constraints) *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 pre-order computation and a post-order computation. The pre-order computation at v essentially performs the computation of Procedure topDown in Fig. 4(b). One important difference is that unlike Procedure topDown which assumes that qualifiers have already been computed (Fig. 4(b), line 7), here we need to introduce variables for the values of the yet undetermined qualifiers.

Example 4.1: Consider the XPath query of Example 2.1. The pre-order computation of the query over the leftmost *client* node of fragment F_0 (shown in Fig. 3(a)) results in the SVect(Q) vector $SV_{client} =< qz_1, 0, 0 >$. Here variable qz_1 indicates that although the node label is indeed *client*, the qualifier for the node is yet to be determined. Contrast this with Example 3.3 that, at the same node, Algorithm PaX3 results in vector $< 1, 0, 0 >$ since the value of qualifiers has already been computed in Stage 1 by PaX3.

For a more complex example, consider the pre-order 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 3.4), while variable qz_2 is due to the qualifier for the node which is still undetermined. In terms of node *name*, the computation results in $SV_{name} =< 0, 0, z_1 \wedge qz_2 >$ □

The post-order computation at a node v starts once every node in the sub-tree rooted at v is visited (always within a fragment). At that point, the qualifiers have been computed for all the nodes in the sub-tree, through a procedure that is similar in spirit with Stage 1 of PaX3. Given the qualifier values at node v, we can unify some of the variables that have been introduced during the pre-order computation.

Example 4.2: Continuing with our last example, after we traverse the sub-tree rooted at the leftmost node *client* of fragment F_0, the qualifiers for the sub-tree rooted at *client* have been computed and are shown in Example 3.1. Given these qualifiers, and since the *client* node is associated only with the first qualifier (entry q_4 of QV_{client}), we can unify variable qz_1 to true. This yields vector $SV_{client} =< 1, 0, 0 >$, the same vector that Algorithm PaX3 computes but only after two passes.

For fragment F_1, the qualifiers for the sub-tree rooted at *broker* are shown in Example 3.2. Since node *broker* is associated only with the second qualifier (entry q_9 of QV_{broker}), we can unify variable qz_2 to y_8. Then, SV_{broker} is now $< 0, z_1 \wedge y_8, 0 >$ 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 sent to S_Q the query answers.

Example 4.3: Continuing with our last example, site S_1 holding fragment F_1 receives the following vectors from S_Q:

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

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

Analysis. While the worst-case complexity of algorithms PaX3 is the same as its PaX2 counterpart, algorithm PaX2 requires *one less* visit. As will be seen in Section 6, our experimental results demonstrate that PaX2 outperforms PaX3.

5. Optimizing Query Evaluation

We now present an optimization that identifies fragments which do not contain any nodes that are in the query answer. The optimization is used by both PaX3 and PaX2 to rule out the identified fragments from any further 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, Fig. 6 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. Note that the additional XPath-annotation requirement imposes negligible space overhead for the fragment tree $\mathcal{F}_T$.

To see how XPath-annotation can help during query evaluation, consider a query Q and a top-down evaluation of

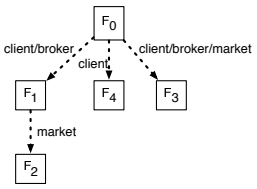


Figure 6: XPath-annotated fragment tree

the selection path of Q. Both algorithms PaX3 (in its Stage 2) and PaX2 (in its Stage 1) perform an evaluation of this type. We propose to first use a top-down evaluation of the selection path of Q over the XPath-annotated fragment tree. Intuitively, performing such an evaluation results in a set of nodes which 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 5.1: Consider a simple query *client/name* over tree T which returns the names of all clients. Evaluating this query over the fragment tree of Fig. 6 finds fragments F_0 and F_4. 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_4 is considered since the procedure knows that a client subtree is included in F_4, 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_3 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/market* from F_0 to F_2 and F_3 does not satisfy the query and therefore fragments F_2 and F_3 are both ruled out. □

XPath-annotations are used before the beginning of Stage 2 in PaX3 and before Stage 1 in 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 our algorithms. In more detail, 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 and 1, respectively, of the algorithms are real answers to the query and can be sent back to site S_Q. Recall from Section 3.2 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 shown in Section 6, XPath-annotations are effective in reducing query evaluation time.

6. Experimental Study

We next present our experimental results. For our experiments we used ten Linux machines (fragment sites), distributed over a local LAN. Each machine has a 3GHz CPU and 1GB of memory. Our datasets consist of trees whose root node is called *"sites"* and each child node of the root

$Q1$	*/sites/site/people/person*
$Q2$	*/sites/site/open_auctions//annotation*
$Q3$	*/sites/site/people/person[profile/age > 20 ∧ /address/country="US"]/creditcard*
$Q4$	*/sites//people/person[/profile/age >20 ∧ /address/country="US"]/creditcard*

Figure 7: Sample queries

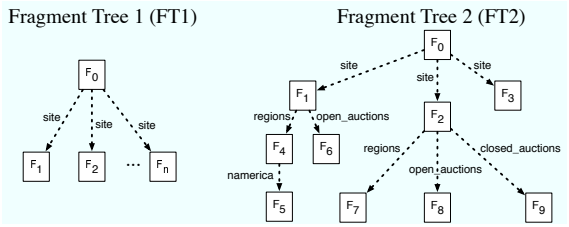
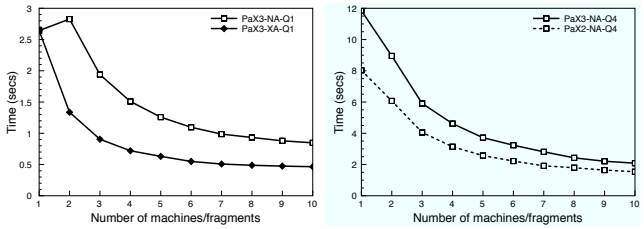


Figure 8: (Annotated) fragment trees used

node is a whole XMark [25] *"site"*. We generated multiple XMark *"sites"* and in each experiment we assigned (fragments of) XMark *"sites"* to different machines. In terms of queries, Fig. 7 shows a sample of the executed queries over our fragmented tree. The choice of presented queries will become clear shortly. In all experiments, reported times are averaged over multiple runs of each experiment. 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 algorithm 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, while if XPath-annotations (XA) are used the line is plotted using the black diamond symbol.

Experiment 1: The objective of this series of experiments is twofold. First, we want to illustrate the benefits of fragmentation. Second, we want to verify the effectiveness and scalability (in number of fragments) of PaX3 and PaX2 both with and without XPath-annotations in the fragment tree. We consider a simple fragment tree like FT1, shown to the left of Fig. 8. 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 Fig. 9(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. In general, note that regardless of the presence of XPath-annotations, fragmentation of trees is beneficial since as fragmentation increases query evaluation time decreases, due to parallelism. The figure also val-



(a) Query $Q1$ (b) Query $Q4$

Figure 9: Evaluation vs. Fragmentation

idates the theoretical analysis of the previous section since the evaluation (parallel computation) cost of PaX3 depends only on the maximum fragment size. Since 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 approx. 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), algorithm 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 returned to the user. 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.

Note that by using the XPath-annotations, the evaluation time of PaX3 is almost halved. The XPath-annotations are used in PaX3 to determine already at the second stage whether a candidate answer is a real answer. Therefore, we save the cost of Stage 3, which is now skipped not just in the first iteration but in all the subsequent ones.

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 Fig. 9(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. XPath-annotations in the fragment tree do not alter the evaluation times of $Q4$ in either of the two algorithms. This is due to the '//' in the selection part of query $Q4$ which, given the fragmentation in FT1, requires us to consider all the fragments.

Experiment 2: The objective of this series of experiments is to study the scalability of our algorithms in terms of query evaluation times, as we increase the size of data. Here we consider a more *natural* fragment tree for our data, shown to the right in Fig. 8. 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 table below 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.

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 different fragments. The increase is such that the cumulative size of the 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.

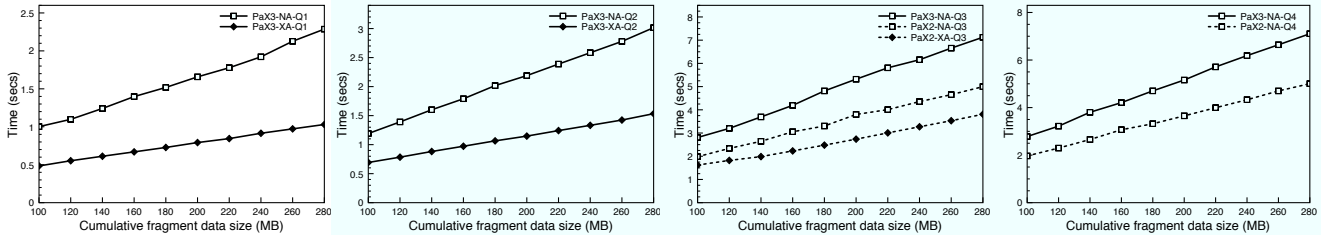
Figure 10(a) clearly shows that algorithm PaX3 scales linearly for query $Q1$, as the data size increases (with or without XPath annotations). The running times for PaX2 are almost identical with the two lines from PaX3, and therefore are not shown in the figure. As explained earlier, this is because both algorithms execute two passes over each fragment, due to lack of qualifiers. The figure also illustrates that evaluation times are more than halved from using XPath-annotations during query evaluation. Specifically, due to the annotations in FT2, the evaluation only considers the data in fragments F_0, F_1, F_2 and F_3. Figure 10(b) shows that the situation is similar, even in the presence of a '//'. Here in spite of the '//' 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 Fig. 10(c). Again, PaX3 scales linearly and 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 therefore any gains made from XPath-annotations are minor, for 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 Fig. 10(c) shows that PaX2 scales also linearly and is faster than PaX3, illustrating once more 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.

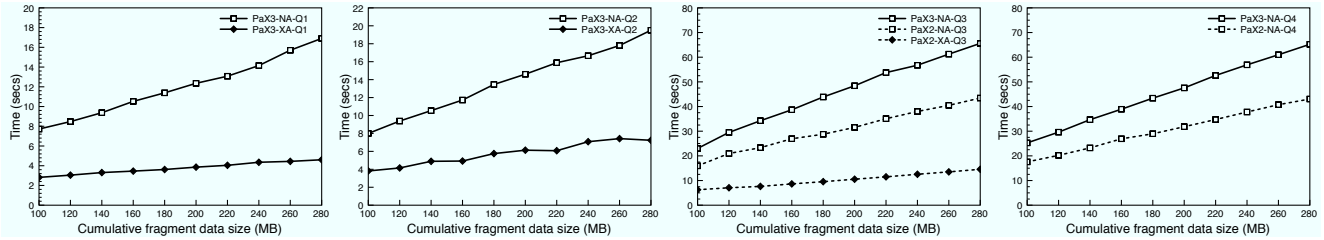
The last query considered, query $Q4$, has a '//' in its selection path and given the fragmentation of FT2, we must evaluate the query (and its qualifiers) over all the fragments of FT2. Here, XPath-annotations do not help in ruling out any fragments. Therefore, as the figure shows, the only gains in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

Experiment 3: This series of experiments are to show that our optimization reduces in practice not only the parallel computation cost of our algorithms, but equally importantly, it also reduces the total computation cost. This experiment builds on the results of Experiment 2 and uses exactly the same setting. To compute the total computation cost of our algorithms, we sum the evaluation times for each machine holding a fragment. Figure 11 shows the total



(a) Query $Q1$ (b) Query $Q2$ (c) Query $Q3$ (d) Query $Q4$

Figure 10: Evaluation time vs. Data scalability, for different queries



(a) Query $Q1$ (b) Query $Q2$ (c) Query $Q3$ (d) Query $Q4$

Figure 11: Total computation time for different queries

computation cost for each query and algorithm of Fig. 10.

Consider Fig. 10(a) and 11(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 11(a) shows that, in addition, the total computation cost was reduced by *two-thirds*. This is because the machines holding fragments that were termed irrelevant to the query did not perform any computation. Therefore, by using XPath-annotations not only the query 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 11(c) and 11(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 Fig. 10(c) and 11(c)), the savings in total computation are again even more significant from those in parallel computation.

Concluding remarks: We have shown that distributing tree fragments over various sites proves an effective strategy with significant reductions in evaluation times. Obviously, always using the (optimized) algorithm PaX2 along with XPath-annotations in the fragment tree 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 a '//' in the selection path of a query, XPath-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 above 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 times.

7. Related Work

Closest to this work is the ParBoX algorithm proposed in [5], which possesses performance guarantees on the total network traffic, computation and communication steps (once). While this work is an extension of [5], the new technical development presented here is substantial and non-trivial. As remarked in Section 1, ParBoX is restricted to *Boolean XPath queries*, which return a single truth value of constant size, and thus the partial answers in their setting are easy to characterize. Indeed, ParBoX is a special case of Stage 1 of the PaX3 algorithm (Section 3.1). In contrast, this work deals with *data-selecting* XPath queries, which return *a set* of XML elements with variable sizes depending on the selectivity of the queries. A direct application of the partial evaluation technique or a direct extension of the ParBoX algorithm will lead to shipping a superset of the answer of a query from *each* participating site, and thus to excessive network traffic that in the worst case is as large as the entire tree rather than the answer of the query. The focus of this work is to figure out what appropriate partial answers should be used, identify *precisely* elements in the answer of the query, and ship *only* those in the final answer to the coordinator site. Despite that this works tackles a far more challenging problem, it achieves the same performance guarantee as its Boolean-query counterpart.

Close to this work are also [27, 2], both with nice performance bounds on total network traffic, computation and communication steps. While [27] studies distributed (data-selecting) query evaluation on semistructured data, [2] provides algorithms for evaluating (aggregate and Boolean) queries on hierarchical distributed catalogs [26]. This work deals with a different problem, namely, (data-selecting) XPath evaluation on XML data. It also differs from [27, 2] in technical approaches. The algorithms of [27, 2] employ query decomposition and focus on query plan generation, which rewrite an input query into sub-queries appropriate for individual sites (using, *e.g.,* the accessibility information of the distributed data); in contrast, this work avoids this overhead by sending *the whole* query to each relevant site; in addition, the algorithms in this paper characterize partial

answers as *Boolean expressions* rather than concrete data as found in [27, 2]. Moreover, an algorithm was given in [19] for evaluating XPath queries on tree fragments distributed in disk pages, employing a nice notion of partial path instances. It differs from this work in the following: (a) it is based on query decomposition and scheduling of the execution of sub-queries, instead of evaluating the same query at all sites and taking expressions as partial answers; (b) it considers a simpler set of XPath queries without qualifiers.

There has been a large body of work on distributed query processing (see, *e.g.,* [21] for a nice survey). The key issue is minimizing communication cost [21]. Based on partial evaluation, this work minimizes both data movement and communication steps by shipping residual functions (Boolean formula) rather than data. Existing techniques for distributed query processing can benefit our algorithms, notably hybrid shipping, two-phase optimization [21], replication [1], parallel query evaluation [8, 14] and mutant query plans [22]. Partial evaluation can also be combined with recent techniques developed for P2P systems (*e.g.,* [7, 13, 17, 9]) and be applied to P2P query processing.

A number of algorithms have been developed for evaluating XPath queries in centralized systems (*e.g.,* [11, 20]). As remarked in Section 1, these algorithms may not work well in a distributed setting, and in addition, the partial evaluation technique may improve the performance of processing XML queries on large XML documents stored in secondary storage in a centralized system. In particular, since partial evaluation of XML queries conducts XML *tree traversal*, the most time consuming part, *in parallel*, it suggests potential optimizations for XML query processing in native XML stores by exploring parallelism. On the other hand, XPath optimizations (*e.g.,* [6, 23]) are complementary to this work.

Partial evaluation has been proven useful in a variety of areas including compiler generation, code optimization and dataflow evaluation (see [18]). Its relevance to query evaluation has surfaced from time to time, most notably in the Disco system [28] and in query rewriting with views and deductive databases [10, 12].

8. Conclusions

We have developed algorithms and optimizations for evaluating generic XPath queries on arbitrarily fragmented and distributed XML trees. We have shown both analytically and experimentally that our techniques are scalable and efficient for handling complex XPath queries on large datasets. We remark that it is far more challenging to partially evaluate data-selecting queries than Boolean queries, and our techniques are among the first for distributed processing of data-selecting XPath queries with *performance guarantees*.

The first topic for future work is naturally the application of partial evaluation to processing XML updates and more expressive XML queries in distributed systems. The second topic, as remarked earlier, is to use the technique to evaluate XML queries on large XML documents in native XML stores. A third topic is to integrate partial evaluation with other optimization techniques for distributed query processing.

Acknowledgment. Wenfei Fan is supported in part by EPSRC GR/S63205/01, GR/T27433/01, and BBSRC BB/D006473/1. The work was done when Gao Cong was at the University of Edinburgh.

9. References

[1] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic XML documents with distribution and replication. In *SIGMOD*, 2003.

[2] S. Amer-Yahia, D. Srivastava, and D. Suciu. Distributed evaluation of network directory queries. *TKDE*, 16(4):474–486, 2004.

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

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

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

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

[7] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying peer-to-peer networks using P-Trees. In *WebDB*, 2004.

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

[9] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online balancing of range-partitioned data with applications to Peer-to-Peer systems. In *VLDB*, 2004.

[10] P. Godfrey and J. Gryz. A strategy for partial evaluation of views. In *Intelligent Information Systems*, 2000.

[11] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *VLDB*, 2002.

[12] A. Gupta, Y. Sagiv, J. D. Ullman, and J. Widom. Constraint checking with partial information. In *PODS*, 1994.

[13] A. Y. Halevy, Z. G. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov. The Piazza peer data management system. *TKDE*, 16(7):787–798, 2004.

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

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

[16] H. V. Jagadish, L. V. S. Lakshmanan, T. Milo, D. Srivastava, and D. Vista. Querying network directories. In *SIGMOD*, 1999.

[17] H. V. Jagadish, B. C. Ooi, and Q. H. Vu. Baton: A balanced tree structure for peer-to-peer networks. In *SIGMOD*, 2005.

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

[19] C.-C. Kanne, M. Brantner, and G. Moerkotte. Cost-sensitive reordering of navigational primitives. In *SIGMOD*, 2005.

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

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

[22] V. Papadimos and D. Maier. Distributed queries without distributed state. In *WebDB*, 2002.

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

[24] Saxon. http://saxon.sourceforge.net/.

[25] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *VLDB*, 2002.

[26] M. Smith and T. A. Howes. *LDAP : Programming Directory-Enabled Apps*. Sams, 1997.

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

[28] A. Tomasic, L. Raschid, and P. Valduriez. Scaling heterogeneous databases and the design of Disco. In *ICDCS*, pages 449–457, 1996.

[29] Xerces and Xalan. http://xalan.apache.org.