XML materialized views in P2P networks

Ioana Manolescu, Spyros Zoupanos
INRIA Saclay–Île-de-France, 4 rue Jacques Monod, 91893 Orsay Cedex, France
firstname.lastname@inria.fr

ABSTRACT

We consider the efficient, scalable management of XML documents in structured peer-to-peer networks based on distributed hash table (DHT) indices. We present an approach for exploiting indices (or materialized views) deployed in the P2P network independently by the peers, to answer an interesting dialect of tree pattern queries. We describe the query (and view) language, provide a rewriting algorithm, discuss view definition indexing strategies based on the DHT, and compare their performance through a set of experiments on a completely deployed platform.

1. INTRODUCTION

The development of electronic document formats has lead to the creation of large warehouses of structured documents, stored and automatically processed to some extent by companies and other organizations (e.g. government bodies etc.). XML data management systems have become more performant over the years, and are currently able to handle well volumes of data such as can be hosted in a single computer. Larger data volumes, or handling documents produced by distributed parties, wishing to retain control over their data, requires the usage of distributed architectures.

In this paper, we describe our approach for building extensible, scalable, flexible *XML warehouses* in a *distributed, peer-to-peer* (P2P) architecture. To be able to provide performance guarantees, we consider *structured P2P* networks, more specifically those based on a distributed hash table (*DHT*) [10] index. Each document resides at a specific peer. Each peer may choose to materialize a set of views, described in a tree pattern language. To disseminate in the network knowledge about the view, we index *view definitions* in the DHT. Each peer may pose queries, expressed in the same tree pattern language as the views. The processing chain for a query q thus becomes: *lookup* the view definitions pertinent for q, say $\mathcal{V}$; *rewrite* q using the $\mathcal{V}$ and pick a rewriting, say r; *execute* r to obtain the results of q.

The problem addressed in this paper can thus be recast as: establishing an architecture, and algorithms, for answering a flavor of XML tree pattern queries using materialized views in a DHT setting.

The contributions of this paper are as follows. (1) We describe a scalable and generic architecture for managing and exploiting materialized XML views in the DHT. (2) We present a new tree pattern query rewriting algorithm based on views, which can be used regardless of the setting (distributed or not). (3) We study several view definition DHT indexing strategies in our architecture. (4) We report on the performance of a complete functional implementation of our proposed architecture and algorithms.

2. ARCHITECTURE AND OUTLINE

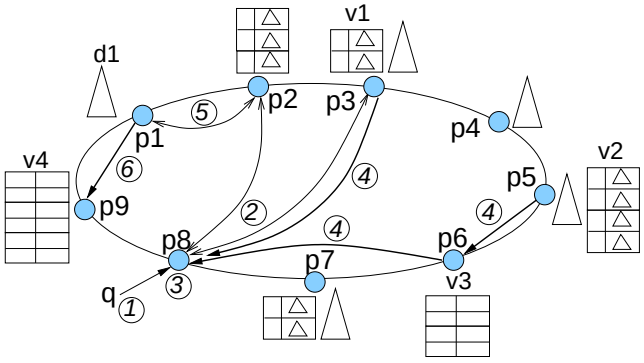

Figure 1: Architecture overview.

The architecture we envision is depicted in Figure 1. Network peers labeled p_1 to p_9 store documents, shown as triangles, and/or views, shown as tables. Such tables may feature attributes of type xml (whose values are serialized XML subtrees), shown as triangles inside tuples. We designate a document d or view v at peer p by the notation $d@p$, respectively, $v@p$.

The tree pattern defining every view is indexed in the DHT. Query processing can be traced following the numbered arrows in the Figure. Assume query q is asked at peer p_8 (step 1). Then, p_8 will perform a DHT look-up to find which view definitions may be useful to rewrite the query. For instance, p_2 and p_3 may return to p_8 relevant view definitions (step 2). Peer p_8 will then run a view-based query rewriting algorithm, trying to reformulate q based on these definitions (step 3). A query rewriting is a logical algebraic plan based on some views, in our example, $v_1@p_3$, $v_2@p_5$, and $v_3@p_6$. After picking a rewriting, p_8 transforms it into a distributed physical plan, which runs in a distributed fashion (step 4, thick arrows denote data transfer). In our example, v_2 is sent to p_6 which joins it with v_3 and sends the result to p_8. At p_8 it joins with v_1 which is sent from p_3.

In the above, we assume each view v is complete, i.e. it includes $v(d)$ for any document d in the network. To obtain such views, whenever a new document, say $d_1@p_1$ in Figure 1, is published, the publishing peer performs another type of lookup (step 5) to determine (possibly a superset of) the view definitions to which the new document may contribute tuples, i.e. such that $v(d_1) \neq \emptyset$. In the Figure 1, such definitions are returned by p_2, and p_1 finds out that d_1 contributes some tuples to the view $v_4@p_9$. The tuples are sent to p_9 (step 6), which adds them to the view extent.

Outline The remainder of the paper is organized as follows. Our pattern language is discussed in Section 3. Section 4 presents the algebra used for rewritings, and Section 5 describes the rewriting algorithm. How views are materialized, indexed in the P2P network, and looked up is discussed in Section 6. We present our experiments in Section 7, discuss related works in Section 8, then conclude.

3. PATTERNS

We will rely on a tree pattern dialect $\mathcal{P}$, defined as follows.

(1) Pattern nodes can correspond to *XML internal nodes* (elements or attributes), or to *leaves* (attribute values, or words in text occurring inside XML elements). For presentation purposes, we do not distinguish between elements and attributes. Observe that we allow a simple word to make up a pattern node, which corresponds to the importance that keyword searches play in our context. We extend the XPath descendant axis to consider that words are descendants of their closest element or attribute ancestors (and extend this descendant relationship via transitivity in the normal way).

(2) Each internal pattern node carries an XML node label from a tag alphabet $A_l = \{a, b, c, \ldots\}$. Each leaf node carries a word label from a word alphabet $A_w = \{\underline{a}, \underline{b}, \underline{c} \ldots\}$.

(3) All pattern edges correspond to the descendant (//) relationships between nodes. We consider that in a large-scale, decentralized scenario, the distinction between child and descendant relationships is not crucial (although it could be easily added).

(4) Each node may be annotated with zero or more among the following labels: id, standing for *structural ID*[1]; $cont$, standing for the full XML subtree rooted at the node; and val, standing for the concatenation of all text descendants of the node, in document order. An id, $cont$ or val label attached to a node denotes the fact that the structural ID, the content or the value, respectively, of the node belong to the pattern result. We assume that node identifiers contain sufficient information to also identify the document to which the nodes belong.

(5) Each internal node may be annotated with a predicate of the form $[val = \underline{c}]$ where $\underline{c} \in A_w$. Such a predicate denotes the fact that data in the view is restricted to nodes on which this predicate restriction applies.

We introduce a simple text syntax for our patterns. We denote nodes by their A_l or A_w label. The possible id, val and $cont$ labels, as well as possible predicates over val, are shown as indices to the node.

For instance, $a_{id\ cont}$ is a pattern storing the structural ID and the content of all a elements. We use paranthesis to show the nesting of children inside parents, and commas to separate the children of the same pattern node among themselves. For instance, $a(b(c_{id}))$ stores the identifier of elements found on some path matching $//a//b//c$. The pattern $a_{[val=5]}(\underline{b}, c_{id})$ stores the identifiers of all c elements having an a ancestor of value 5, and whose serialized XML subtree contains the word $\underline{b}$.

Let p be a pattern and d be an XML document. As customary, an embedding $\phi : p \rightarrow d$ of p in d is a function associating d nodes to p nodes, preserving node labels and ancestor-descendant relationships. The result of evaluating p on d, denoted $p(d)$, is the set of tuples obtained by lining together in a tuple, all structural IDs and/or values and/or serialized content, for each embedding of p in d. Extending this to a set $\mathcal{D}$ of documents, the semantics of evaluating p over $\mathcal{D}$ is defined as $\cup_{d \in \mathcal{D}} p(d)$ which also identifies the document to which the nodes belong.

We say two patterns p_1, p_2 are *equivalent*, denoted $p_1 \equiv p_2$, if for any database $\mathcal{D}$, $p_1(\mathcal{D}) = p_2(\mathcal{D})$. The $\mathcal{P}$ pattern dialect we use is closely related to XPath$^{\{//,[]\}}$ [15], and indeed the polynomial-time containment decision algorithm of [15] can be adapted to $\mathcal{P}$ patterns. Thus, $\mathcal{P}$ pattern equivalence can be established in polynomial time in the size of the patterns.

4. ALGEBRAIC QUERY REWRITING

Let $q \in \mathcal{P}$ be a query and $\mathcal{V} = \{v_1, v_2, \ldots, v_k\}$ a set of views, $v_i \in \mathcal{P}, 1 \leq i \leq k$. A *rewriting of q using $\mathcal{V}$* is an expression $e(v_1, v_2, \ldots, v_k)$, belonging to a logical algebra $\mathcal{A}$ (which we describe next), such that $e \equiv q$, that is, for any database $\mathcal{D}$, $e(v_1(\mathcal{D}), v_2(\mathcal{D}), \ldots, v_k(\mathcal{D})) = q(\mathcal{D})$.

Let v be a view and $op, op' \in \mathcal{A}$ be algebraic expressions (or plans, in short). For explanation purposes, whenever a pattern stores either the identifier or the value or the content of a node labeled l, we will use either $l.id$ or $l.val$ or $l.cont$, respectively, as the name of the corresponding attribute. The following plans belong to $\mathcal{A}$:

(1) $scan(v)$ (or v, in short) is a scan of all v tuples.

(2) $\pi_{pList}(op)$ is a projection over op, where $pList$ is the list of columns to keep.

(3) $\sigma_{cond}(op)$ is a selection over op, where $cond$ is a conjunction of predicates of the form $i \odot \underline{c}$ or $i \odot j$, where i, j are op attribute names, $\underline{c} \in A_w$, and $\odot \in \{=, \prec\}$ is a binary operator. We use $\prec$ to designate the "is ancestor of" predicate. Thus, assuming the attributes named i and j contain structural identifiers, $\sigma_{i \prec j}(op)$ returns those op tuples where the identifier in attribute i corresponds to an ancestor of the node whose identifier is in attribute j.

(4) $nav(op, i, np)$ is a navigation operator, applying the navigation described by the pattern np over the i attribute of op. Two restrictions apply. First, i must be a $cont$ attribute, as navigation only makes sense inside an XML fragment. Second, np nodes must not have id indices, since it is generally not possible to determine the structural identifier corresponding to a node by only considering a *fragment* enclosing the node. For example, let $v = a_{id\ cont}$. The operator $nav(scan(v), a.cont, b(c_{cont}, d_{cont}))$ returns tuples with 4 attributes: a identifiers, a contents, and contents of c and d descendants of a, having a common b ancestor below a.

(5) $op \bowtie_{pred} op'$ is a join operator, where $pred$ is a predicate as for selections.

(6) $fetch(op, i)$, where op's attribute i contains element identifiers, adds to each op tuple an extra column, containing the serialized ($cont$) representation of the XML subtree rooted in the node whose identifier was in column i in that tuple.

To properly define the rewriting problem we face, we need a few auxiliary notions.

Two plans $a_1, a_2 \in \mathcal{A}$ are *algebra-equivalent* if a_2 can be obtained from a_1 via: usual rewriting rules from the relational algebra (e.g. pushing selections and projections, join re-ordering etc.); transitive closure of ancestor-descendant predicates; or pattern compo-

[1] Structural identifiers are unique node identifiers that allow to decide only by comparing the identifiers of two nodes n_1 and n_2, whether n_1 is an ancestor of n_2 or not. Popular examples include (pre-order, post-order) labeling [3], Dewey IDs [19] etc.

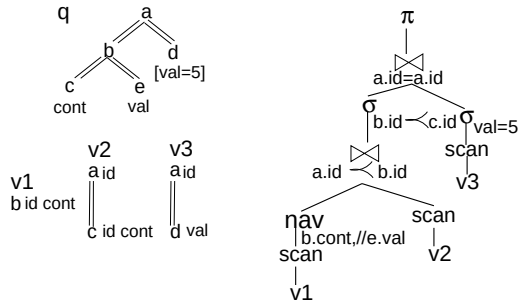

Figure 2: Sample query, views, and rewriting.

sition. We illustrate the last two items via examples. The plan $\sigma_{a.id \prec c.id \wedge b.id \prec c.id}(op_1 \bowtie_{a.id \prec b.id} op_2)$ is algebraically equivalent to $\sigma_{b.id \prec c.id}(op_1 \bowtie_{a.id \prec b.id} op_2)$, given that the predicate $a.id \prec c.id$ is implied by $a.id \prec b.id$ and $b.id \prec c.id$. Finally, let $p_a = a_{cont}$, $p_b = b_{cont}$, $p_c = c_{cont}$, and $p_{bc} = b_{cont}(c_{cont})$. Then, $\pi_{c.cont}(nav(scan(p_a), a.cont, p_{bc})) \equiv \pi_{c.cont}(nav(nav(scan(p_a), a.cont, p_b), b.cont, p_c))$. Intuitively, the left-hand plan navigates directly from a to its descendants on the path $//b//c$, whereas the right-hand plan navigates first to the b descendants (inner nav) and then from those, to their c descendants (outer nav).

An algebraic plan $a \in \mathcal{A}$ is *minimal* if there exists no sub-expression a' of a, such that $a \equiv a'$. As an example, let $p_a = a_{id}$, $p_b = b_{id}$, and $p_c = c_{id}$, and consider $\alpha = \pi_{a.id}((scan(p_a) \bowtie_{a.id \prec b.id} scan(p_b)) \bowtie_{a.id=a.id} (scan(p_a) \bowtie_{a.id \prec c.id} scan(p_c)))$ Expression α returns the identifiers of all a elements having b and c descendants. Then, $\alpha \equiv \alpha'$, where $\alpha' = \pi_{a.id}((scan(p_a) \bowtie_{a.id \prec b.id} scan(p_b)) \bowtie_{a.id \prec c.id} scan(p_c))$ is a sub-expression of α. Therefore, α is not minimal; α' is.

Query rewriting problem Given a set of views $\mathcal{V}$ and a query q, the problem of rewriting q based on $\mathcal{V}$ consists of finding the set of all minimal equivalent $\mathcal{A}$ rewritings of q, up to algebraic equivalence.

Figure 2 provides an example. The only rewriting of the query q based on the views v_1 to v_3 is the plan shown in the Figure. Observe that no view contained a node labeled e, but we extract it from the $b.cont$ attribute via navigation. To properly combine v_1 and v_2, the b node of v_1 is "pushed" under the a node of v_1 by the leftmost join, but an extra selection is needed to specify that it is above the c node.

On patterns, algebra, and rewriting We make here some useful remarks concerning our pattern language $\mathcal{P}$ and our algebra language $\mathcal{A}$. First, observe that $\mathcal{P}$ is conjunctive - intuitively, no $\mathcal{P}$ pattern can refer e.g. to all the a nodes *lacking* b descendants. As a consequence (and similarly to the context of rewriting conjunctive Datalog queries with conjunctive views [14]), adding union ($\cup$) to $\mathcal{A}$ does not increase the set of equivalent rewritings for a given $\mathcal{P}$ query q.

Second, observe that for any expression $e_1 \in \mathcal{A}$ and database D, if $e_1(D) = \emptyset$, then $e_2(e_1(D)) = \emptyset$ for any $\mathcal{A}$ expression e_2 built on e_1. To see why this is true, notice that each of the algebraic operators *op*, applied on an empty input $\emptyset$, returns $\emptyset$.

Finally, we notice an important property concerning the *usefulness* of a view $v \in \mathcal{P}$ in rewriting a given query $q \in \mathcal{P}$:

($*$) If v can be used to rewrite q, then there exists a label- and structure-preserving embedding $\phi : v \rightarrow q$.

Algorithm 1: **DPR**: dynamic programming query rewriting

Input : query pattern q, view patterns $v_1, \ldots, v_n$
Output: minimal algebraic rewritings of q using $v_1, \ldots, v_n$

foreach $nq \in q$ **do**
  $P_1(nq) = \emptyset$
foreach $v_i \in \{v_1, \ldots, v_n\}$ **do**
  foreach *embedding* $\phi : v_i \rightarrow q$ **do**
    $ppp' \leftarrow$ new $ppp(scan(v_i), v_i)$
    $ppp'' \leftarrow$ extend(ppp')
    handlePPP(ppp'', 1)
$j = 2$
repeat
  $P_j(n) \leftarrow \emptyset$ for all $n \in q$
  foreach $n \in q$ **do**
    foreach $ppp_j^n \in P_{j-1}(n)$, $ppp_1^n \in P_1(n)$ **do**
      $ppp_2 \leftarrow ppp_{j-1}^n \bowtie_{n.id=n.id} ppp_1^n$
      handlePPP(ppp_2, j)
    foreach $m \in q$, *m is a child of* n *in* q **do**
      foreach $ppp_{j-1}^n \in P_{j-1}(n)$, $ppp_1^m \in P_1(m)$ **do**
        $ppp_3 \leftarrow ppp_{j-1}^n \bowtie_{n.id \prec m.id} ppp_1^m$
        handlePPP(ppp_3, j)
  $j \leftarrow j + 1$
until $\forall n \in q, P_j(n) = \emptyset$

Proof: Assume on the contrary that a rewriting of q based on v exists, and that v cannot be embedded into q. Let d_q be the XML document obtained by copying each q node into an XML element, preserving q structure. Clearly, $q(d_q) \neq \emptyset$. Since v cannot be embedded in q, by definition of view contents, $v(d_q) = \emptyset$. Let $D_q = \{d_q\}$ be the database consisting of exactly d_q. We have $q(D_q) \neq \emptyset$, whereas for any algebraic expression e, in virtue of the second observation above, $e(v(D_q)) = \emptyset$. Thus, we can exhibit a database D_q such that $e(v(D_q)) \neq q(D_q)$ for any algebraic expression e. Therefore, no equivalent rewriting of q can be built based on v, which contradicts our hypothesis.

5. REWRITING-BASED QUERY ANSWERING

In this section, we describe how queries can be answered in our architecture, based on materialized tree pattern views.

We first describe two algorithms (Sections 5.1 and 5.2) for finding algebraic rewritings of a query q based on a set $\mathcal{V}$ of views. Our algorithms are based on a generate-and-test paradigm: we build increasingly larger algebraic expressions, and test them for equivalence to the query. Such tests are complicated by the difference of language: rewritings are $\mathcal{A}$ expressions, whereas the query belongs to P. To overcome this, rewriting manipulates *(plan, pattern) pairs* (or *ppp*s, in short), that is, pairs consisting of an algebraic plan and an *equivalent* tree pattern. This equivalence is established from the start: rewriting starts with one *ppp* for every view v, where the pattern is v and the plan is $scan(v)$. As larger *ppp*s are constructed, equivalence is ensured between the pattern and the pair.

Our rewriting algorithms are detailed in Sections 5.1 and 5.2, and some of their properties are considered in Section 5.3. We then briefly explain (Section 5.4) how logical rewritings are tranformed into executable plans.

5.1 Dynamic programming algorithm (DPR)

The first algorithm we consider follows a dynamic programming approach (Algorithm 1), and therefore we term it **DPR** for dynamic programming rewriting. DPR organizes the *ppp*s enumerated during the search in a family of sets using a family of parameterized *ppp* sets $P_j(nq)$, where:

- nq is a query node, and for any j, all the *ppp* in $P_j(nq)$ have at least one node which could be embedded in nq (via a label- and structure- preserving embedding as in the previous section);
- j is an integer, equal to the number of joins in the algebraic plan of any *ppp* in $P_j(nq)$ plus one. Thus, for any $nq \in q$, plans in $P_1(nq)$ are built directly from a single view v_i; the algebraic plan of any *ppp* in $P_2(nq)$ is a join over two views, the plans of *ppp*s in $P_3(nq)$ is a join over three views etc.

Populating the initial *ppp* sets The first step in the algorithm is to populate the sets P_1 (lines 3-7). We first compute, for each view v_i, all possible embeddings of v_i in q, and create a pair ppp' for each embedding $\phi : v_i \rightarrow q$, in order to reflect all the ways in which v_i could be used to rewrite the query (line 5).

A subtlety arises now: our simple embeddings do not take into account node attributes, which may lead to missing some opportunities for rewriting. For instance, consider a view a_{cont} and a query $a(b_{cont})$. Clearly, the query can be rewritten using the view, but for this, we need to inject the information that the view stores the full a-rooted subtrees in the rewriting process. To that effect, we extend ppp' by (i) adding nodes to its pattern, and (ii) adding nav operators on top of the plan, whenever a v_i node storing $cont$ is embedded into a non-leaf q node (line 6). We now outline the role of the **extend** function (for which we do not show pseudo-code). Let nv be a v_i node labeled l_{cont}, and $\phi(nv) = nq$. Let $desc(nq)$ be the forest obtained by copying nq's children subtrees, stopping above nodes carrying the id label. Extending ppp' by navigation produces a new pattern, ppp'', which is a copy of the one in ppp' but (i) adds $desc(nq)$ as children of nv in the pattern and (ii) adds $nav(..., nv.cont, desc(nq))$ on top of the equivalent plan.

We illustrate the extension of ppp' for the view v_1 and the query q in Figure 2. The initial *ppp* is $ppp' = (scan(v_1), v_1)$. The single b node of v_1 is embedded into the b node of q. By extension, we obtain the pair $(nav(scan(v_1), b.cont, (c_{cont}, e_{val})),\ b_{cont}(c_{cont}, e_{val}))$, which we denote ppp_b in the sequel. The query nodes covered by ppp_b are those labeled b, c and e.

Observe that in ppp'', all possible extensions via navigation are applied. Thus, we navigate from b both to find c descendants, and to find e descendants. However, the rewriting in Figure 2 only uses the e_{val} extension of v_1's b node, and uses v_2 to obtain the c query node. The extra navigation to c_{cont} will be recognized as unnecessary, and removed, prior to producing solutions (see below).

Extended patterns are inserted in all the sets $P_1(nq)$ to which they belong, by calling the function **handlePPP**, shown in Algorithm 2. This function checks whether the incoming *ppp* can be turned into a solution after some adjustments, i.e. selections and projections. Adjustment also eliminates useless navigations which had been eagerly added at line 6 in Algorithm 1.

Algorithm 2 then checks (line 5) if the algebraic plan in *ppp* plan is new, i.e. not algebraically equivalent to the plan of any other previously explored *ppp*. This test is important for two reasons. First, recall that we are interested in solutions up to algebraic equivalence only. Second, if a newly found *ppp*'s plan is algebraically equivalent to a plan already explored, it is important that we detect this *early on*, to avoid the time-consuming useless search based on the new *ppp*. How testing is performed will be explained and better understood shortly below.

Algorithm 2: (Plan, pattern) pair handling

Input : (plan, pattern) pair *ppp*, layer index j
Output: none (side effects on the set $P_j(n)$ and on the solution set S)

```
1 ppp2 ← adjust(ppp, q)
2 if ppp2.pattern ≡ q then
3     add ppp2.plan to S
4 else
5     if ppp2 is a new partial solution for q then
6         foreach l ∈ q such that a ppp2.pattern node can be
          embedded in l do
7             add ppp2 to Pj(l)
```

In the example of Figure 2, lines 1-7 of Algorithm 1 produce the following sets: $P_1(a) = \{(scan(v_2), v_2),\ (scan(v_3), v_3)\}$; $P_1(b) = \{ppp_b\}$, where ppp_b is obtained by navigation as explained above; $P_1(c) = \{(scan(v_2), v_2),\ ppp_b\}$; $P_1(d) = \{(scan(v_3), v_3)\}$; and $P_1(e) = \{ppp_b\}$.

Combining (plan, pattern) pairs The second part of Algorithm 1 develops increasingly larger plans and patterns, progressing towards rewritings. We proceed in layers, indexed by the layer counter j, starting from $j = 2$. In each layer, we add (plan, pattern) pairs in the sets $P_j(nq)$, $nq \in q$, by building node ID equality joins and structural joins out of the (plan, pattern) pairs contained in the sets P_{j-1} and P_1. We use one *ppp* from P_1 in all joins, in order to develop only left-deep join plans, since rewriting needs to enumerate all plans only up to algebraic equivalence. Thus, the equivalent bushy join plans (and their *ppp*s) do not need to be developed.

We now trace this on the example in Figure 2, starting on the sets P_1 enumerated above.

For $\boxed{j = 2}$, we obtain:

1. For $n = a$, considering equi-joins on $a.id$ (line 13 in Algorithm 1), we add to $P_2(a)$, $P_2(c)$ and $P_2(d)$ the (plan, pattern) pair $(scan(v_2) \bowtie_{a.id=a.id} scan(v_3),\ a_{id}(c_{id,cont},\ d_{val}))$, which we will denote ppp_j^1 in the sequel.
2. For $n = a$ and $m = b$, we join $(scan(v_2), v_2)$ with ppp_b on $a.id \prec b.id$ and obtain the pair ppp_j^2 described by:
   - the plan $scan(v_2) \bowtie_{a.id \prec b.id} nav(scan(v_1), b.cont, (c_{cont}, e_{val}))$
   - the pattern $a_{id}(b_{id,cont}(c_{cont}, e_{val}),\ c_{id,cont})$

In this pattern, one c node is a child of the b node and the other is a child of the a node. To better fit ppp_j^2 to the query, the **adjust** function adds a selection ensuring that the second c node is a child of the b node, turning ppp_j^2 into ppp_j^3:

- the plan $\sigma_{b.id \prec c.id}(scan(v_2) \bowtie_{a.id \prec b.id} nav(scan(v_1), b_{cont}, (c_{cont}, e_{val})))$
- the pattern $a_{id}(b_{id,cont}(c_{cont}, c_{id,cont}, e_{val}))$

We add ppp_j^3 to the relevant sets, namely: $P_2(a)$, $P_2(b)$, $P_2(c)$ and $P_2(e)$.

3. Still for $n = a$ and $m = b$, we join (v_3, v_3) with p_b on $a.id \prec c.id$ and obtain the pair ppp_j^4 consisting of:
   - the plan $scan(v_3) \bowtie_{a.id \prec b.id} nav(scan(v_1), b_{cont}, (c_{cont}, e_{val}))$
   - the pattern $a_{id}(b_{id,cont}(c_{cont}, e_{val}),\ d_{val})$

ppp_j^4 is added to the sets $P_2(a)$, $P_2(b)$, $P_2(c)$, $P_2(d)$ and $P_2(e)$.

At this point for $j = 2$ we have $P_2(a) = \{ppp_j^1, ppp_j^3, ppp_j^4\}$, $P_2(b) = \{ppp_j^3, ppp_j^4\}$, $P_2(c) = \{ppp_j^1, ppp_j^3, ppp_j^4\}$, $P_2(d) = \{ppp_j^1, ppp_j^4\}$ and $P_2(e) = \{ppp_j^3, ppp_j^4\}$.

For $\boxed{j = 3}$, for $n = a$, we combine $ppp_j^3 \in P_2(a)$ with $(scan(v_3), v_3) \in P_1(a)$ and obtain the pair ppp_j^5 having:

- the plan $scan(v_3) \bowtie_{a.id=a.id} \sigma_{b.id \prec c.id}(scan(v_2) \bowtie_{a.id \prec b.id} nav(\ldots))$
- the pattern $a_{id}(b_{id,cont}(c_{cont}, c_{id,cont}, e_{val}), d_{id})$

where $nav(\ldots)$ is the nav plan of ppp_b.

Signatures for efficient equivalence checks We now discuss checking that the algebraic plan of a newly found *ppp* is not algebraically equivalent to any previously found plan (line 5 in Algorithm 2). One could test the plan directly against the whole set of existing plans. However, this would be inefficient even for small plan sets, since it requires enumerating many rewriting rules (such as join re-ordering) to see which plan may turn into another etc.

Instead, we assign to each (plan, pattern) pair a *signature*, intuitively describing "what view nodes does it use to cover each query node", or "*how* does it cover query nodes". More precisely, as explained above, each node n_p in the pattern of the pair is obtained from either (i) one view node, or (ii) an ID equality join over several view nodes. By construction, the pattern from the pair can be embedded in the query; let n_q be the query node to which n_p embeds. In case (i), the pair signature will associate to n_q the single view node. In case (ii), the pair signature will associate to n_q the set of view nodes joined by ID equality.

For example, consider again the (plan, pattern) pair ppp_b. Its signature is: $\{b \rightarrow \{b_{v1}\}; c \rightarrow \{c_{ext,v_1}\}; e \rightarrow \{e_{ext,v_1}\}\}$, where b_{v_1} is the b node in v_1, and c_{ext,v_1}, e_{ext,v_1} are the nodes obtained by extending v_1, navigating downwards from the b node. Now, consider the pair ppp_j^1, obtained by joining v_2 and v_3 on their a nodes. Its signature is: $\{a \rightarrow \{a_{v_2}, a_{v_3}\}; c \rightarrow \{c_{v_3}\}; d \rightarrow \{d_{v_3}\}\}$.

The following property holds for *ppp*s and their signatures:

($\circ$) Let ppp_1, ppp_2 be two (plan, pattern) pairs and s_1, s_2 be their signatures. The plans of ppp_1 and ppp_2 are algebraically equivalent *iff* $s_1 = s_2$.

Based on this property, a set of explored signatures is maintained during the search, and a new *ppp* is added to some $P_j(nq)$ set only if its signature was not previously explored. This resembles dynamic programming-based query optimization, where only one join plan for a given set of relation occurences is kept [16].

5.2 Depth-first search algorithm (DFR)

Algorithm DPR, as all dynamic programming algorithms, tends to produce solutions towards the end of the search only. If there are many views to start with, this means the rewriting time may be prohibitive. Therefore, we also develop an alternative rewriting algorithm **DFR**, based on depth-first search. DFR differs from the DPR in the way it organizes and explores its *ppp*s.

Instead of the sets $P_j(nq)$, DFR uses a family of sets $C_i(nq)$, where nq is a query node, and i indicates how many query nodes are covered by a *ppp* in C_i. Thus, if a view v of 5 nodes was embedded in the query, the pair $(scan(v), v)$ will be added to the sets $C_5(n_q)$ for each query node to which some v node was embedded. During plan enumeration, DFR always attempts to combine: a *ppp* from a set $C_i(n_q)$ of the largest possible i value; and a *ppp* of the form $(scan(v), v)$ for some view v. Thus, similarly to DPR, DFR only enumerates left-deep plans; but, differently from DPR, it builds on the plan covering the largest number of query nodes, regardless of how many views that plan uses. Both algorithms use *ppp* signatures to detect equivalent plans.

The trade-offs between DPR and DFR are the following. DFR often finds *some* rewritings much faster than DPR, since it greedily progresses towards rewritings covering as many query nodes as possible. In exchange, the total DFR search time is larger than DPR's, due to the "disordered" fashion in which DPR explores the search space. In practice, given that one typically needs only one (or a few) rewritings, if the set of views is very large, DFR (constrained to stop after a few rewritings or at a timeout) is preferrable.

5.3 Remarks on the rewriting algorithms

We make here some remarks on the two rewriting algorithms. Their *termination* is due to the fact that we only explore *ppp*s whose *minimized* pattern can be embedded in the query, and there is only a finite number of such patterns. Patterns are systematically minimized when creating new *ppp*s (details omitted for lack of space; the algorithm is closely inspired from [4]).

The algorithms' *correctness* relies on the fact that the plan and the pattern of a *ppp* are always equivalent. Hence, when the pattern is found equivalent to the query (line 2 of Algorithm 2), the plan is a rewriting. The equivalence among the plan and the pattern of a *ppp*s is established from the start, and is maintained at every step when two *ppp*s are combined into a larger one. The **adjust** function plays a role here, as illustrated in the previous section by the transformation of ppp_j^2 into ppp_j^3. The algorithms' *completeness* is ensured by their generate-and-test approach, and by proposition ($\circ$), which ensures we only disregard plans for which an equivalent one has already been explored.

Regarding *efficiency*, both algorithms only consider views that can be embedded in the query, which helps trim down useless parts of the search space. Finally, one can question their *effectiveness*, or: how can we choose a *good* rewriting among the possible ones, or, if we stop the search after finding just a few solutions, how do we ensure those are good? To answer this, we first have to choose a measure for rewriting quality. While the real execution time for a given rewriting depends on the physical plan that can be devised to implement the logical rewriting plan, we found reasonable to consider a rewriting is good if it uses *few* views, on the basis that many-views rewritings entail joins, which may be expensive. The rewriting using the least views can be found by exhaustive search. However, we cannot ensure finding it *early on* during the search. A k-optimal solution can perhaps be found along the lines of [2]; we will consider that as part of our future work.

5.4 After rewriting

A given logical rewriting must be transformed into a physical (executable) query plan in order to produce results. This step corresponds to physical query optimization, and is performed by a dedicated module present on each DHT peer. Since the views involved in the rewriting may reside on different peers, the physical plan is a distributed one. The tuple-oriented execution engine includes operators such as view scan, selection, projection, hash join, sort, send and receive to handle data transfers, and two physical structural join operators, StackTreeDesc and StackTreeAnc [3].

Clearly, many physical plans can be used to implement a given logical one, and distribution only increases the search space. Our optimizer applies standard heuristics for choosing the peer on which to place each operator [20] in order to reduce the search space. The plan retained is the one that minimizes (i) the number of inter-peer data transfers, and among those with least transfers, (ii) the number of sort operators. In the best case, if all the joins of a rewriting

plan are structural, a fully pipelined (non-blocking) physical plan is generally found.

6. P2P VIEW MANAGEMENT

We have so far explained how to exploit views for query rewriting. In this section, we consider how views are materialized (Section 6.1), and identified in order to rewrite a query (Section 6.2) in the DHT network. Both operations require some *view definition indexing* in the DHT. We stress that we do not index view extent (tuples), but only the pattern defining the views.

We start by introducing a useful term: if d is a document and v is a view such that $v(d) \neq \emptyset$, we say d *affects* v.

6.1 View materialization

Assume peer p decides to establish a view v. Then, when a peer p_d publishes a document d affecting v, p_d needs to find out that v exists. To that effect, view definitions are *indexed for document-driven lookup* as follows. For any label (node name or word) appearing in the definition of the views $v_1, v_2, \ldots, v_k$, the DHT will contain a pair where the key is the label, and the value is the set of view URLs $v_1, v_2, \ldots, v_k$.

When a peer p_d publishes a document d, p_d performs a lookup with all d labels (node names or words) in order to find a superset S_a of the views that d might affect. Subsequently, p_d evaluates $v(d)$ for each $v \in S_a$. We implemented this step based on a SAX traversal, with time complexity in $\Theta(|d| \times |v|)$. In practice, large fragments of d are typically not interesting for a given view v, thus computing $v(d)$ tends to spend some time traversing useless parts of d. To share this cost, we group view definitions in batches of some size n (we set $n = 10$) and evaluate all the views of a batch in a single d traversal. Thus, d fragments useless to all batch views are parsed only once per batch.

Finally, p_d sends, for each $v \in S_a$ such that $v(d) \neq \emptyset$, the tuple set $v(d)$ to the peer p_v publishing v. For performance, tuples are encoded so that the identifier of d (a rather lengthy URL) is sent only once, even if it appears (possibly several times) in each tuple of $v(d)$ (recall that element IDs contain sufficient information to identify the document they belong to). The decoding is reversed on arrival at p_v, prior to storing the tuples.

We have so far considered that v is published before the documents affecting it. The opposite may also happen, i.e. when v is published, a document d affecting v may already exist, and $v(d)$ needs to be added to v's extent. To that effect, we require the publisher p_d of a document d to periodically look up the set of views potentially affected by d, and send $v(d)$ to those views as described above. Thus, v will be up to date (reflecting all network documents that affect it) after the periodical check and subsequent actions have been performed by all document publishing peers.

We end the section by considering view maintenance in the face of document deletion or change. When documents are deleted from the system, a similar view lookup is performed, and the peers holding the views are notified to remove the respective data. We model document changes as deletions followed by insertions.

6.2 Identifying views for rewriting

A second form of view definition indexing is performed in order to enable finding views that may be helpful for rewriting a given query. In this context, a given algorithm for extracting (key, value) pairs out of a view definition is termed a *view indexing strategy*. For each such strategy, a *view lookup* method is needed, in order to identify, given a query q, (a superset of) the views which could be used to rewrite q. Many strategies can be devised. We present two that we have implemented, together with the space complexity of the view indexing strategy, and the number of lookups required by the view lookup method.

Label indexing (LI): index v by each v node label (either some element name $a \in A_l$ or some constant $\underline{a} \in A_w$). The number of (key, value) pairs thus obtained is in $O(|v|)$.

View lookup for LI: look up by all node labels of q. The number of lookups is $\Theta(|q|)$.

The LI strategy coincides with the view definition indexing for document-driven lookup (described in the previous section). Its drawback is its lack of precision. For instance, a view $a_{ID}(c_{ID})$ will be retrieved for all queries involving the terms a, although it is useless for all queries not containing c. A more precise strategy is the following.

Leaf path indexing (LPI): let $LP(v)$ be the set of all the distinct root-to-leaf label paths of v. Index v using each element of $LP(v)$ as key. The number of (key, value) pairs thus obtained is in $\Theta(|LP(v)|)$.

View lookup for LPI: let $LP(q)$ be the set of all the distinct root-to-leaf label paths of q. Let $SP(q)$ be the set of all non-empty sub-paths of some path from $LP(q)$, i.e., each path from $SP(q)$ is obtained by erasing some labels from a path in $LP(q)$. Use each element in $SP(q)$ as lookup key.

As an example, let $v = a_{ID}(b_{ID}, c_{ID})$, then v will be indexed by the keys $a(b)$ and $a(c)$. Let q be the query $a(f(b_{ID}, c_{ID}))$. With LPI, the view lookups will be on a, $a(f)$, $a(b)$, $a(c)$, f, $f(b)$, $f(c)$, b, and c. Thus, v will (correctly) be identified as potentially useful to rewrite q. Indeed, if a second view $v' = f_{ID}$ exists, then $q = \sigma_{f \prec b \wedge f \prec c}(v \bowtie_{a \prec f} v')$.

Let $h(q)$ be the height of q and $l(q)$ be the number of leaves in q. The number of lookups is bound by $\Sigma_{p \in LP(q)} 2^{|p|} \leq l(q) \times 2^{h(q)}$.

The view lookup strategies described above can be shown to be complete, thanks to Proposition $(*)$ (Section 4).

7. PERFORMANCE EVALUATION

We have fully implemented the platform described so far. We used Java 6 for the implementation.

We describe preliminary experiments carried on a cluster of 10 PCs with Intel Xeon 5140 CPU @ 2.33GHz and 4GB of Ram. Berkeley DB [7] (version 3.2.76) and FreePastry [11] (version 2.1 alpha) are used for storing view data and indexing view definitions respectively. For the experiments we use 30 peers which are distributed uniformly to all the computers of our cluster. Together, the peers publish a set of 100 XMark benchmark documents [17], and 30 views. All the documents affect all the views. Adding unrelated views and documents do not impact the performance of query processing, which is the main focus of our experiment.

The testing scenario is as follows. First, the views are published and indexed in the DHT network using the LI strategy, and the views are filled with data. A query is asked at one of the peers, which performs all necessary steps in order to process the query and return answers. For the graphs that follow, we use 7 sets of 100 documents of increasing data size. The sizes of the sets begin from 11MB and extended to 1.6GB. For every set we perform the same experiment 4 times in order to get more accurate average values.

7.1 View building

In Figure 3 we can observe the total time that the peers devoted in extracting tuples from the documents for all the views. Recall that these measurements are summed up for all the peers and may even be artificially high because in fact views are materialized in parallel. What is interesting to notice in this diagram is that the extraction time is linear compared to the total size of the documents, something that was also described in Section 6.1.

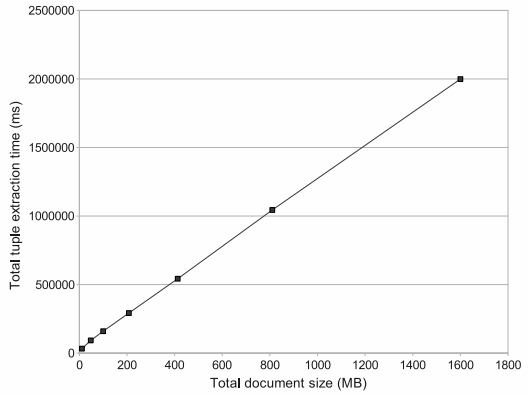

Figure 3: Total tuple extraction time for sets of documents of different sizes.

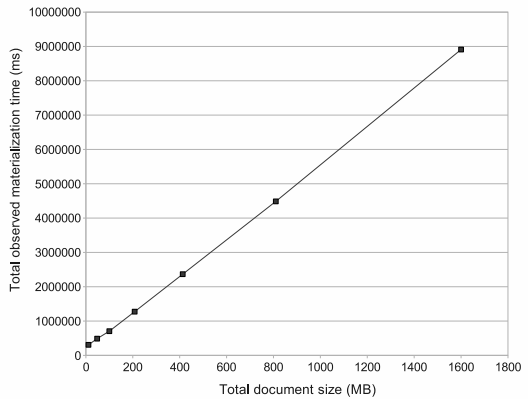

Figure 4: Total observed view materialization latency for sets of documents of different sizes.

After extracting the necessary tuples for a view from a document, the next step to be done is to send them and to store them at the peer that is responsible for that specific view. We call the time needed to perform this action as "observed materialization latency" and Figure 4 shows how this value is affected by the total document size.

7.2 Query processing

The query processing includes rewriting the query using a set of views that produces a logical plan, optimizing a logical plan for execution that produced a physical plan and, in the end, the execution of the latter. In this experimental section we will consider the query $site_{id}(regions_{id}(africa_{id}(item_{id})), catgraph_{id}(edge_{id}))$.

Query rewriting & optimization These two steps are performed in memory by a single peer, and are extremely fast when compared to full execution (of course regardless of the database size). More specifically, in the 30 view experiment that we analyze, the rewriting needed on average 50.68 ms and the optimization 13.82 ms. The query rewriting module used the DSR algorithm, restricted to develop at most 10 rewritings.

Query execution You can observe in Figure 6 how much time a physical plan needs to be executed for different document set sizes. The plan that we execute is shown at Figure 5 and it joins 2 views from which one of them is at the peer where we ask the query and the other one is at a different peer.

Such plans are quite representative of our scenario because our purpose is to established materialized views that are likely to cover large parts of a query (or cover the query completely).

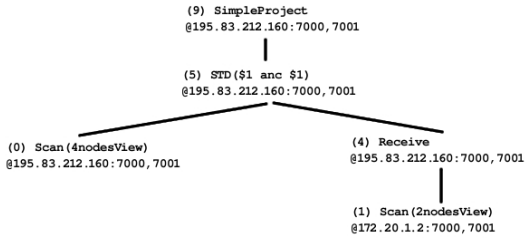

Figure 5: Physical plan of the query executed in the experiments.

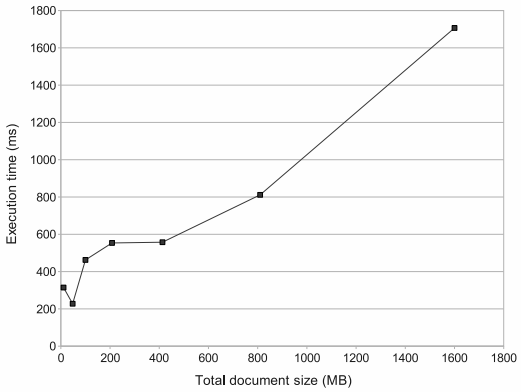

Figure 6: Execution time for increasing data size.

Table 1 shows the number of results that we get for sets of different sizes. In Figure 7 we can notice that the execution time is linear to the number of results obtained for our query. We should point out that in 1.7 seconds we get 1.16 milions of results which is 680 results per millisecond.

Total document set size	# of results
11	109
48	600
100	4500
208	18000
413	72000
810	288000
1600	1160000

Table 1: Result size for the different sets of documents.

7.3 Evaluation outcome

In summary our experiments have shown that query execution and view materialization scale up linearly with the total data size in the scenario we considered. While limited, we consider these results encouraging for the overall platform behavior. Moreover query rewriting and optimization are reasonably fast demonstrating that the overhead of views may be acceptable.

8. RELATED WORKS

Our work follows a series of approaches for handling large volumes of XML documents in structured P2P networks, based on DHTs [12, 8, 18, 1]. In these works, the focus is on indexing *documents* in the DHT so that XML queries can be processed fast.

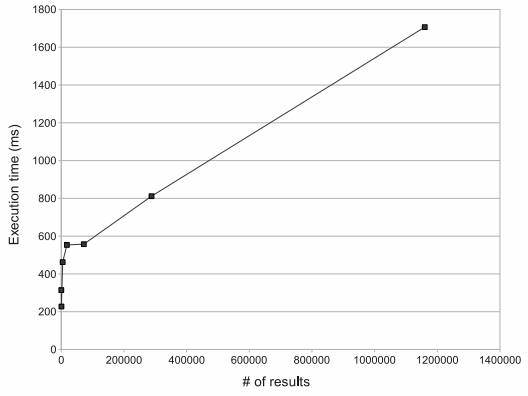

Figure 7: Execution time and result number.

In contrast, we focused on establishing and exploiting *views* over the whole set of XML documents in the DHT, and exploiting these views for query processing. The work described here can be seen as a generalization of [1], which established a view of the form l_{ID} for any label (node or keyword) l appearing in some document. The interest of the views we propose is that they can be established by peers interested in improving the performance of a given set of queries in the style of a local query cache, and they also allow "re-publishing" (re-packaging) XML content, e.g. in order to structure a topic-specific portal hosted by one peer or set of peers.

Our DPR rewriting algorithm is related to the one described in [6], which rewrites queries expressed in a richer XAM formalism [5] over XAM views, under structural constraints encapsulated in a Dataguide [13]. The work described in [6] does not consider distribution. In contrast, in our setting, given that documents and views are published independently by different distribtuted parties, we felt that the precise XAM features (nesting, optional edges, ...) created fine distinctions that most users of our system would not be willing to consider. Therefore, we rely on a simpler tree pattern model in this work. Moreover, the Dataguide used in [6] to rewrite queries impacts query rewriting at all levels, making it further different from the DPR algorithm we describe, which does not require such constraints.

9. CONCLUSION

We have considered the problem of building and exploiting materialized XML views in a DHT network where peers independently publish XML documents and/or materialized views. We have proposed an architecture and algorithms for building and maintaining the views, as well as for processing peer queries based on the existing views in the DHT network. All the described algorithms have been fully implemented in a functional Java-based platform, on which we currently carry out experiments.

Many avenues for future work exist. The most important one for now is to exploit the trade-offs between various view indexing strategies for query processing and for view materialization, between space taken up by the view indexing pairs and time consumed looking up views and processing extra view definitions that are potentially not useful. We are also interested in improving our physical optimizer so that it identifies quickly efficient distributed query plans, and in experimenting with holistic structural twig joins [9], which we have used successfully in [1].

10. ACKNOWLEDGMENTS

We thank Alin Gabriel Tilea for his help in developing, maintaining and testing the VIP2P platform. Part of the VIP2P code originates from the ULoad platform and has been developed by Andrei Arion.

This work has been partially funded by *Agence Nationale de la Recherche*, decision ANR-08-DEFIS-004.

11. REFERENCES

[1] S. Abiteboul, I. Manolescu, N.Polyzotis, N. Preda, and C. Sun. XML processing in DHT networks. In *ICDE*, 2008.

[2] F. N. Afrati, C. Li, and J. D. Ullman. Generating efficient plans for queries using views. In *SIGMOD*, 2001.

[3] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, 2002.

[4] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Tree pattern query minimization. *VLDB J.*, 11(4), 2002.

[5] A. Arion, V. Benzaken, and I. Manolescu. XML access modules: Towards physical data independence in XML databases. In *XIME-P*, 2005.

[6] A. Arion, V. Benzaken, I. Manolescu, and Y. Papakonstantinou. Structured materialized views for XML queries. In *VLDB*, pages 87–98, 2007.

[7] Oracle Berkeley DB Java Edition. http://www.oracle.com/technology/products/berkeley-db/je/index.html.

[8] A. Bonifati, U. Matrangolo, A. Cuzzocrea, and M. Jain. XPath lookup queries in P2P networks. In *WIDM*, 2004.

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

[10] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common API for structured P2P overlays. In *Proc. of IPTPS*, 2003.

[11] Freepastry, an open-source implementation of pastry. http://freepastry.org/FreePastry/.

[12] L. Galanis, Y. Wang, S. Jeffery, and D. DeWitt. Locating data sources in large distributed systems. In *VLDB*, 2003.

[13] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, 1997.

[14] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogeneous information sources using source descriptions. In *VLDB*, 1996.

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

[16] J. G. Raghu Ramakrishnan. *Database management systems.* McGraw-Hill Professional, 2003.

[17] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. Xmark: A benchmark for xml data management. In *VLDB*, 2002.

[18] G. Skobeltsyn, M. Hauswirth, and K. Aberer. Efficient processing of XPath queries with structured overlay networks. In *OTM Conferences (2)*, 2005.

[19] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang. Storing and querying ordered xml using a relational database system. In *SIGMOD Conference*, pages 204–215, 2002.

[20] A. Tomasic, L. Raschid, and P. Valduriez. Scaling heterogeneous databases and the design of disco. In *ICDCS*, 1996.