HAL
open science

Adaptive distributed XML views

Asterios Katsifodimos, Ioana Manolescu, Alin Tilea, Spyros Zoupanos

To cite this version:

Asterios Katsifodimos, Ioana Manolescu, Alin Tilea, Spyros Zoupanos. Adaptive distributed XML views. 2009. inria-00433481

HAL Id: inria-00433481

https://inria.hal.science/inria-00433481v1

Preprint submitted on 19 Nov 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Adaptive distributed XML views

Asterios Katsifodimos Ioana Manolescu
Spyros Zoupanos
INRIA Saclay – Île-de-France and LRI, Université de Paris Sud-11
firstname.lastname@inria.fr

4 November 2009

Abstract

We consider the problem of efficiently managing large corpora of XML documents in a distributed, decentralized setting involving hundreds of machines (or peers). We assume distributed users are willing to share their data. Thus, on any network peer, users may establish long-running subscriptions, by means of declarative queries over the entire current and future data. Thus, XML data is "liquid": it comes from many sources and flows towards its consumers. Moreover, data accumulating at a peer as a result of a subscription is seen as a materialized view over the whole network, and can be used to answer other queries.
We have developed ViP2P, a fully functional system implementing these ideas. We describe the architecture and the algorithms behind it, as well as experiments on hundreds of machines distributed in a LAN, in [7]. In this report, we consider the issue of *adapting* the materialized views established by a given ViP2P peer, to the needs of the peer itself, and of the other peers in the network. We outline the goals and techniques of adaptation, and discuss practical algorithm for evolving the set of views of a peer to better suit the needs of the peer, and of the whole network.

1 Overview

We consider a distributed, decentralized setting, where independent peers may publish XML documents to be shared with other peers of the network. Sharing takes the form of enabling all peers to query documents published on any peer, without having to know which document is published where.

To enable one peer to discover interesting documents, a distributed XML index is needed. Such an index is updated whenever a document is published, removed, or updated; examples include [1, 4]. For better performance, a peer may also establish a *materialized view*, that is, a query to be evaluated over all the documents of the network (past, present and future), and whose results are stored at the peer. Such materialized views allow redundancy, and provide a basis for adapting the distributed store to the queries asked in the network.

We have designed and built ViP2P (standing for *V*iews in *P*eer-to-*P*eer, a platform enabling peers to share XML documents by means of materialized views. Any ViP2P peer can publish XML documents and/or views described by conjunctive XML tree pattern queries. Once a view is published:

- the view definition is *indexed* in a distributed fashion over the network;
- the view is *maintained*, that is, documents published before or after the view will contribute data to the view extent. The view definition index is used by the publisher of a document to learn which views may stand to receive data from the document.
- views may be used to *rewrite* subsequent queries. The view definition index is used by the peer where a query is asked, to determine which views may be used to rewrite the query.

The architecture and algorithms for view indexing, maintenance and view-based query rewriting have been described in a separate paper [7]. Importantly, in that work, we considered the set of views on each peer to be given and immutable.

In this paper, we consider an important complementary problem: how to chose the materialized views to be established by each peer. Since in a data-sharing network the data and query loads vary in time, the set of views must also change with time. Therefore, we discuss *adaptation strategies* which may change the set of views of a peer, to improve the performance of the peer's queries and that of queries to which the peer contributes.

This paper is organized as follows. Section 2 presents the language we use to defined views and queries, and the different classes of materialized views supported in ViP2P. Section 3 recalls the basic strategy of indexing and maintaining views, and re-using them to rewrite queries, in order to make this paper self-contained. Section 4 discusses view adaptation.

2 ViP2P materialized views

2.1 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* (words in text occurring inside XML elements, or in attribute values). For presentation purposes, we do not distinguish between elements and attributes. We extend the XPath descendant axis to consider that words are children of their closest element or attribute ancestors. Each internal pattern node carries a label from a tag alphabet $A_l = \{a, b, c, \ldots\}$. Each leaf node carries a label from a word alphabet $A_w = \{\underline{a}, \underline{b}, \underline{c} \ldots\}$.
2. Pattern edges correspond to parent-child or ancestor-descendant relationships between nodes.

3. Each pattern node may be annotated with some *stored attributes*, describing some information items that the pattern stores out of each XML node matching the pattern node. The *cont* annotation indicates that the full (serialized) image each matching XML tree node is stored. The *id* annotation indicates that a node identifier, which uniquely identifies the node (and the document it belongs to). Moreover, we assume *structural* IDs, i.e. such that one may decide, by comparing the identifiers of two nodes n_1 and n_2, whether n_1 is an ancestor/parent of n_2 or not. Many variants of structural identifiers exist, e.g., [2, 6, 9], some of which provide further information, e.g. allow identifying the least common ancestor of two nodes etc. *For the purpose of this work, we only require that parent-child and ancestor-descendant relationships can be determined from the node IDs.* Finally, the *val* labels stands for the node's text value, obtained by concatenating all its text descendants in document order.

4. Each node may be annotated with a predicate of the form $[val = \underline{c}]$ where $\underline{c} \in A_w$, restricting the XML nodes which match the pattern node, to those satisfying the predicate.

Notations and syntax simplification We say a pattern node *has* an *id*, respectively *val*, *cont*, or value predicate, if the node is decorated with such an index.

We introduce a simple text syntax for patterns. We denote nodes by their A_l or A_w label. The possible *id*, *val* and *cont* labels, and predicates over *val*, are shown as indices to the node. For instance, $a_{id\,cont}$ is a pattern storing the structural IDs and the content of all a elements. We use parenthesis 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 IDs 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 $\underline{b}$.

Pattern semantics 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 [3]. The result of evaluating p on d, denoted $p(d)$, is the list of tuples obtained by lining together in a tuple, all IDs and/or values and/or serialized content, for each embedding of p in d. Assuming a total order over the nodes of p (top-down, left-to-right traversal), the tuple order in $p(d)$ is dictated by the lexicographic order over the d nodes which are targets of the embeddings. For a document set $\mathcal{D}$, the semantics of p over $\mathcal{D}$ is defined as the concatenation (in the order of the document IDs) of all $p(d)$, $d \in \mathcal{D}$.

We use $a.id$ (respectively, $a.val$, $a.cont$) to denote the corresponding attribute in $p(\mathcal{D})$.

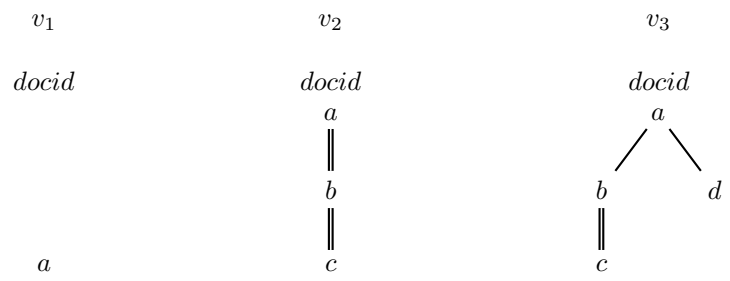


Figure 1: Sample document level views.

2.2 From patterns to views

Based on tree patterns, we are now ready to present ViP2P views. The main idea is that while a pattern may express fine-granularity structural conditions over an XML document, we allow two different granularities for materialized views: either *document level*, or *node level*.

Document level views A document level view v_{dg} includes exactly one tuple for any document d such that $v_{dg} \neq \emptyset$. That tuple contains only the document URI. With respect to notation, we distinguish views at the document level by the fact that they do not have any *id*, *val* or *cont* attribute.

Figure 1 presents 3 examples. View v_1 stores the URIs of all the documents that may have a node labeled a. View v_2 stores the URIs of all documents that have an a node which has a descendant b node, which has a descendant a c node. More generally, any tree pattern previously described, stripped of all its stored attributes, can become a document level view.

Node-level views A node level view is defined by any pattern $p \in \mathcal{P}$, and stores the result of evaluating $p(d)$ over all documents $d \in \mathcal{D}$ ($\mathcal{D}$ is the set of documents published in the ViP2P platform).

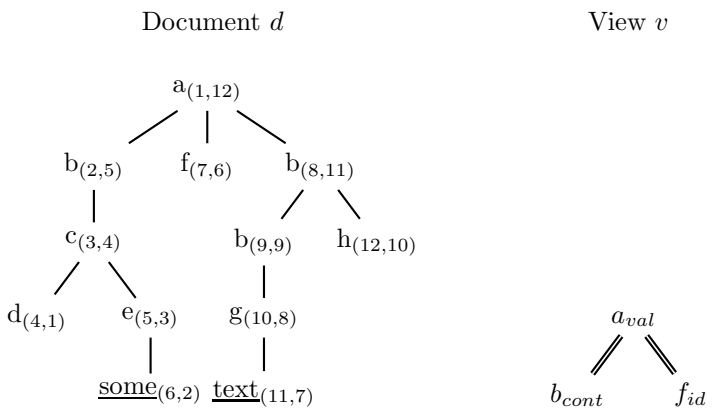
Figure 2 shows a sample document, a node lavel view and the result of evaluating that view over that document.

3 Outline of ViP2P processing

To make this paper self-contained, we briefly recall from [7] how views are maintained and advertised in ViP2P (Section 3.1) and how queries are processed (Section 3.2. Concerning this last aspect, we explain how to process queries involving document-level views. Such views were not considered in [7].

3.1 View management

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*



Result of evaluating $v(d)$

a_{val}	b_{cont}	f_{id}
some text	<c><d/><e>some</e></c>	(7,6)
some text	<g>text</g><h/>	(7,6)
some text	<g>text</g>	(7,6)

Figure 2: A document, a node level view and the result of evaluating the view against the document.

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) to find a superset S_a of the views that d might affect. Then, p_d evaluates $v(d)$ for each $v \in S_a$. Further optimizations are discussed in [7]. Finally, p_d sends, for each view v, the tuple set $v(d)$ (if it is not empty) to the peer p_v publishing v. Furthermore, p adds (p_v, v) to a list $contribTo(d)$ of all the peers storing views to which d has contributed.

We have so far considered that v is published before the documents which affect 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.

When a peer deletes a document, messages are sent to the peers in $contrib2(d)$, instructing them to delete the tuples of d.

3.2 Query processing

When a query q is asked on a peer p_q, p first performs a look-up to identify definitions of views which may be used to rewrite q. It retrieves a set of views, and explores alternative rewritings. A rewriting is retained, which is a logical plan combining a set of views, say, $v_1@p_1$, $v_2@p_2 \ldots, v_k@p_k$. The next steps depend on the levels of the views $v_1, \ldots, v_k$.

Node level views only If all the views $v_1@p_1, v_2@p_2, \ldots, v_k@p_k$ are at node level, the rewriting is then transformed (optimized) into a physical executable plan, which only involves some of peers $p_1, p_2, \ldots, p_k$ and p_q. Thus, the only peers participating to the processing of a query, are peers holding views that contribute to the query rewriting, and the query peer.

For example, consider the query $q = //a//b_{cont}$ and the views $v_1@p_1$ defined as $//a_{id}$ and $v_2@p_2$ defined by $//b_{id,cont}$. A possible rewriting is $r = v_1@p_1 \bowtie_{a.id \prec b.id} v_2@p_2$. The structural join $\bowtie$ could for instance be located at p_2 in the physical plan. Then, p_1 would send its tuples to p_2, which will perform the join, and send the final results to p.

Some document level views If some or all of the views are at document level, processing the rewriting plan only produces a list of document URIs. The respective peers are then contacted and the query is shipped to them for evaluation. Query results are to be sent directly to p.

For example, consider the query $q = //a[//b]//c_{cont}$ and the views $v_1@p_1$ defined as $//a_{id}$ (node level), $v_2@p_2$ defined as $//b$ (document level), and $v_3@p_3$ defined as $//c_{id,cont}$ (node level). A posible plan to rewrite q is:

$$fetch(((v_1@p_1 \bowtie_{a.id \prec c.id} v_3@p_3) \cap_{docID} v_2@p_2), q)$$

In this expression, the $\bowtie$ identifies occurences of the $//a//c$ pattern. We then intersect the document identifiers corresponding to these occurences, with the document identifiers of v_2. We thus obtain the identifiers of documents which certainly feature the $//a//c$ pattern, and some b elements, which may or may not be ancestors of an a with a c descendant. The final operator $fetch(\cdot, q)$ dispatches the query q to all the peers storing documents whose identifiers are provided by its input. These peers will process q and forward the results to the query peer.

Observe that one could have also used a simpler plan:

$$fetch(((\pi_{docID}(v_1@p_1) \cap_{docID} \pi_{docID}(v_3@p_3)) \cap_{docID} v_2@p_2 q), q)$$

where prior to $fetch$, we only intersect the document IDs of the three views. This requires less data transfers from v_1 and v_3, however, it is less precise, as it may return more document identifiers than the previous plans, and thus potentially leads to contacting more peers than needed.

4 View adaptation

In this section, we discuss how the materialized views established on each ViP2P peer are initialized, and how they evolve. Section 4.1 provides the basic assump-

tions we make. Section 4.2 discusses view adaptation.

4.1 Assumptions

First assumption: query availability and compulsory views We make the assumption that at any time, it must be possible to answer any query $\mathcal{P}$ query over the peer network. This obviously assumes (i) some techniques to deal with peer failures (ViP2P uses replication to increase view availability), and (ii) tolerance for the time it takes to propagate the contribution of a newly arrived document to all the views to which it should contribute. We will not discuss these aspects further here. What matters is that there must be a way to process any query expressed in our tree pattern dialect $\mathcal{P}$.

To achieve this, one may broadcast the query to all peers of the network, but this is obviously very costly. Instead, we require that all peers in the network store some *compulsory views*, whose size is kept to a minimum, and which allow answering queries more efficiently. We denote the set of compulsory views of peer p by $p.comp$.

As a consequence, each peer p must have a non-zero *view storage budget*, denoted S_p.

Second assumption: peers' interest and selfish views We furthermore assume that any peer that issues a query is interested in reducing the overall processing time of the query. To that effect, peers may establish *selfish views*, which lead to more efficient execution strategies for the peer. Observe that the definitions of selfish views are indexed in the ViP2P network, too. Thus, they may also benefit other users (and other queries), even though they were mainly established for the interest of the peers holding them. We denote teh set of selfish views of a peer p by $p.self$.

Third assumption: willingness to help and collaborative views Finally, we assume that a peer is willing to devote some space to establish views if only to help the processing of other peers' queries. We denote the set of collaborative views of a peer p by $p.collab$.

Putting it all together, we obtain:

$$S_p = S_{p,comp} + S_{p,self} + S_{p,collab} + S_{p,free}$$

where S_p is the space budget of peer p, $S_{p,comp}$ is the space that p uses to store its compulsory views, $S_{p,self}$ is the space used to store p's selfish views, $S_{p,collab}$ is the space used for collaborative views, and $S_{p,free}$ is the space unused so far at peer p. Ideally, $S_{p,free}$ should be close to zero, to maximize the benefits that the storage space of p brings to this and the other network peers. When a peer joins the network, $S_{p,comp} = S_{p,self} = S_{p,collab} = 0$ since the peer initially has no views. Upon joining, p is assigned some compulsory views and thus thus $S_p = S_{p,comp} + S_{p,free}$.

Data statistics We assume available for each view v, regardless of its type, its number of tuples denoted $|v|$. Moreover, for each document d, we assume available the number of tuples from d that belong to v and we denote this number by $|v(d)|$. The statistics $|v|$ and $|v_d|$ are indexed in the network together with (as part of) the definition (metadata) of the view v. For instance, let v be the view $//a//b_{cont}$, then the definition of v will be indexed in the DHT by the node label a, say, at peer p_{xa}, and by the node label b, say, at peer p_{xb}. Then, p_{xa} stores the association between the key a and the value consisting of the triple:

- $//a//b_{cont}$ (the definition of v)
- $|v|$
- $\forall d \in \mathcal{D}, |v(d)|$

We assume such triple values inserted in the view definition index are sufficiently small not to cause performance problems. Should the set of values $|v(d)|$ become too large, it can be compressed into a histogram structure, trading precision for space in a relatively standard way.

Query statistics We assume a given time interval named *query history window* and denoted hw. On each peer, for each query q that has been asked on the peer p, or to which the peer p contributed, we store:

- $n_{hv}(q)$, the number of times q has been asked during the time interval of length hw which ends at the current moment, $c_{hw}(q)$, the average cost (response time) of q during this time window and $e_{hw}(q)$, the effort spent by the peer in processing q during the same window
- similarly, $n(q)$, $c(q)$ and $e(q)$, respectively the frequency, cost, and effort incurred to p, due to the query q.

Observe that these statistics are related to the peer and are local to the peer in the following sense: if the peer asks no query and is never solicited to participate to processing another peer's queries, the peer has no query statistics.

View utility model Given a view v, we define a utility function $u(v)$ which aims at quantifying the cost of materializing and storing a view as well as the benefit that it would bring to the processing of a given workload (set) of queries $Q = \{q_1, q_2, \ldots, q_n\}$ if it is materialized. The utility is defined by:

$$u(v) = \sum_{i=1}^{n} cost(q_i)|_{\neg v} - \sum_{i=1}^{n} cost(q_i)|_v$$

where, the $cost(q_i)|_{\neg v}$ is the cost of answering a query q_i given that the view v is absent and $cost(q_i)|_v$ is the cost of answering the query q_i in the presence of view v. A high utility value signals a view whose materialization is interesting - from the perspective of the workload Q.

4.2 View set evolution

A peer joins the network with all its space budget free, and no views. Upon joining, he is assigned a set of *compulsory views*. These are all document level views of the form $//a$, where a is the name of some XML element occuring in a $\mathcal{D}$ document, or $//\underline{a}$, where $\underline{a}$ is a keyword occuring in some text node or attribute value of a $\mathcal{D}$ document. A peer p is assigned a view $//a$ such that $h(a) = p$ where h is the hash function of the DHT. We assume that for each peer, the space budget is sufficient to store these compact compulsory views. Compulsory views are re-distributed as peers joins and leave the network as follows. Consider the compulsory view $//a$ such that $h(a) = p_1$ and the peer identifier p_1 is not assigned in the network. The view will then be stored to the peer with the smallest identifier p_2 such that p_2 is larger than p_1 and the identifier p_2 is assigned. If and when the identifier p_1 is assigned later on, the view will be moved to that peer, and its URI accordingly updated in the view index, so that the view may be found at the correct peer.

The views may then evolve in time by applying one of several *adaptation rules*, while obeying the following constraints:

- a compulsory view is never dropped
- peers are more interested in shortening the processing time of their own queries, than in shortening the processing time of other queries
- if free space is available, peers may add collaborative views to shorten the processing of other peers' queries. Such views may be dropped later on in favor of selfish views (selfish views pre-empt space).

The adaptation process is sketched in Algorithm 1. It consists of two main procedures: the first one tries to materialize an extra view, first, for selfish reasons. A view is materialized if its utility for the peer's workload of own queries, Q_{self}, is positive, and greated than the utility of any other possible view. Still, there may be no view recommended at this point, if e.g. the peer has no queries, or they are sufficiently well served by the existing views, or the top recommended view is larger than the available space. In this case, procedure *viewAdapt* will attempt similarly to materialize the best recommendation for the workload Q_{others}.

Procedure *viewDrop* removes all views whose recent usage has fallen below a given threshold.

Finally, procedure *adapt* periodically triggers *viewDrop* (to see if some space can be claimed back) and then *viewDrop*.

View wish list The algorithm outlined above has a shortcoming: empty space may go forever unused on a peer, if that peer issues no query, and it is never solicited to answer other peers' queries, either. In this case, $Q_{self} = Q_{others} = \emptyset$ and Algorithm 1 would not cause any changes.

To cope with such situations, we may allow each peer to build a *wish list* of views that may help its queries, and advertise it over the network. Wish list

```
t procedure viewAdapt begin
    if S_{p,free} > 0 then
        if Q_self ≠ ∅ then
            recV ← recViewFor(Q_self, S_{p,free});
            // recV is a list of positive utility views for
            // the workload Q_self, in decreasing utility order
            if recV ≠ ∅ then
                p.self ← p.self ∪ {recV[0]};
                update p.free
            end
        end
    end
    if p.free > 0 then
        if Q_others ≠ ∅ then
            recV ← recViewFor(Q_others, S_{p,free});
            // recV is a list of positive utility views for
            // the workload Q_self, in decreasing utility order
            if recV ≠ ∅ then
                p.collab ← {recV[0]};
                update S_{p,free}
            end
        end
    end
end

procedure viewDrop begin
    foreach view v ∈ p.self ∪ p.collab do
        if n_hw(v) < n_0 then
            drop v;
            update S_{p,free}
        end
    end
end

procedure adapt begin
    foreach time instant t ∈ {τ_0, 2τ_0, ..., nτ_0, ...} do
        viewDrop;
        viewAdapt;
    end
end
```

Algorithm 1: Procedures used in view materialization

views are indexed just like any other views in ViP2P, but they are marked as virtual and to-be-established. The number of peers having added a given view to their wish list is globally computed. A view having exceded a given treshold of popularity is put in a special shortlist of *most wanted views*, and eventually, adaptation at an empty, unused peer will lead to its materialization.

5 Related works

An interesting recent work is [8], which studies recommending XQuery views based on a set of XQuery queries. Our work is obviously related. At a superficial level, an important difference is the query language, since we consider tree patterns and not XQuery. More importantly, our views contain identifiers, based on which fragments of XML can be joined back into tree patterns, whereas identifiers are not present at the XQuery language level. Moreover, we consider document level views as a way to save space while retaining the important information. Finally, we consider view sharing in a peer-to-peer setting and focus on striking a balance between views established in a selfish purpose and those meant to help other peers process their queries.

In [5], linear XPath views are jointly maintained and exploiting in peer-to-peer settings. The authors consider the situation where a group of peers collaborate to help each others' queries by establishing shared caches. There are many differences between our and that work. First, they only support navigation-based rewriting of a query based on a single view, whereas we are able to support multiple-view rewriting. Second, their work is mostly centered in simple parent-child linear XPath views, which can be organized in Trie structures due to their special simple form. In our case, a query of the form $//a//b$ can be written based on two views $//a_{id}$ and $//b_{id,cont}$. Obviously, there is no way of organizing such view definitions in a single view index tree, although the views may one day be used jointly.

References

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

[2] 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.

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

[4] L. Galanis, Y. Wang, S. R. Jeffery, and D. J. DeWitt. Locating Data Sources in Large Distributed Systems. In *VLDB*, 2003.

[5] K. Lillis and E. Pitoura. Cooperative xpath caching. In *SIGMOD*, pages 327–338, 2008.

[6] J. Lu, T. W. Ling, C. Y. Chan, and T. Chen. From Region Encoding To Extended Dewey: On Efficient Processing of XML Twig Pattern Matching. In *VLDB*, 2005.

[7] I. Manolescu and S. Zoupanos. Materialized views for P2P XML warehousing. *Journées de Bases de Données Avancées* (informal proceedings only), 2009. Available at http://vip2p.saclay.inria.fr/papers/bda09.pdf.

[8] N. Tang, J. X. Yu, H. Tang, M. T. Özsu, and P. A. Boncz. Materialized View Selection in XML Databases. In *DASFAA*, 2009.

[9] 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*, 2002.