P2P Views Over Annotated Documents

Konstantinos Karanasos, Ioana Manolescu

P2P Views Over Annotated Documents

Konstantinos Karanasos, Ioana Manolescu

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 and present an approach for answering queries by exploiting materialized views deployed in the DHT network independently by the peers. We describe how our approach can be employed to also handle RDF-annotated documents and provide algorithms to index and materialize the views in the DHT, as well as rewrite the queries using such views.**

I. Introduction

In recent years, more and more software tools, including the most user-friendly ones, such as text editors, have started to export their contents into some *structured document format*, such as HTML or XML. Moreover, large-scale organizations need to produce, deploy and exploit large volumes of data, and in particular structured XML data. Publication of such resources is inherently distributed. One could consider uploading all published content to a single site. However, this raises some scalability problems, which may require acquiring dedicated hardware, and introduces a single point of failure. In order to deal with such content, we have presented the VIP2P platform [1], which enables efficient management of XML documents on a distribute hash table (or DHT, in short [2]). VIP2P exploits the materialized views deployed on the DHT independently by the peers, to answer an interesting dialect of tree pattern queries.

At the same time, *annotations* have become very popular as a means to add information to a given document. HTML Meta tags, Dublin Core [3] and social networks' tagging are among the most common methods to express annotations. Here, we designate by annotation any simple statement in the style of the RDF standard [4], attaching to a given subject (or resource, such as a document, or a small portion of text) a named property, with a certain value. Using documents *and* annotations provides the flexibility to handle a variety of application scenarios in which documents or RDF alone would not be suitable. As an example, think of a Web page containing a news item, annotated by a human reader or a text analysis tool to point out the person names appearing in the page, her positions within various institutions etc. or to express subjective opinions regarding the document. One could suggest modifying the Web page to incorporate the additional information of the annotations. However, this is not always feasible, since the author of the annotations may be distinct from, and have no control over, the author of the original document; furthermore, the original document should be readable also to those that are not interested in the extra information. Using only RDF to model all the content, on the other hand, is not appropriate since end users are familiar with, and expect to use structured documents.

To extent the capabilities of VIP2P, we have built AnnoVIP, which is capable of handling more complex queries and views and, thus, addresses applications such as the join manipulation of documents and annotations. At the core of content sharing in AnnoVIP stand *materialized views over the whole network content*. Each peer may define views, describing patterns of inter-connected documents and annotations, that the peer is interested in. Once a view is established, its definition will be indexed in the DHT network. When documents or annotations are published, by looking up in the DHT, the publishing peer learns if its new content may contribute to some view, and if yes, it sends the respective data to the view. After publication, this lookup is repeated periodically to identify contributions to views defined later on. Thus, views are updated over time, in the manner of long-running, de-centralized subscriptions.

A further step in content sharing in AnnoVIP is *materialized view-based query rewriting*. Here, we consider the situation when a peer issues an ad-hoc query, which it has not declared as a local view. The peer then looks up in the DHT the existing view definitions, and may rewrite its query based on the views. From a rewriting, a distributed query plan is derived and evaluated. When available, pre-computed materialized views may lead to efficient query evaluation.

The novelty of AnnoVIP stems from its built-in dual support for documents and annotations at arbitrary granularity (one can annotate a document, an element, or even a text fragment). Maintaining and exploiting materialized views for efficient query processing, over such interconnected corpora of documents and annotations requires new algorithms, further complicated by the distribution on the DHT.

This report is organized as follows. Section II discusses the data model of AnnoVIP and Section III presents the query and view language used. View materialization is described in Section IV, whereas view-based query rewriting is presented in Section V. We relate AnnoVIP to existing works in Section VI.

II. Data Model

The basic kind of content we consider consists of XML documents. Each document d published by peer p has an URI allowing to uniquely determine d inside p and in the whole network.

Since we consider handling annotated documents as a very significant application that could be addressed by our system, we have enabled AnnoVIP to target content at very different granularity levels. Thus, one can refer to a document, an element, a text node, or even a fragment of text, e.g., a phrase of particular significance, or a person's name inside a text. Therefore, we consider that any fragment of a document d, whatever its size, has an URI. Such URIs are implemented by (offset, length) pairs identifying the fragment in the serialization of d. Moreover, the URI of d can be easily obtained from the URI of any fragment of d. This holds in many common URI schemes, such as XPointer [5], where $d.URI$ is a prefix of all the URIs of elements in d.

Our model assumes that any XML element has a child labeled *URI*, whose value is the actual URI of the element. However, URI-labeled nodes are virtual, that is, they do not actually appear in the elements (although as we will explain, they are needed for querying documents and annotations). In a similar manner, any fragment can be seen as a node, endowed with an URI.

The second class of content we consider concerns annotations, which may be either produced by human users, possibly with the help of some tools, or by automated modules (e.g. recognizing named entities within a document). For simplicity, we consider that all annotations have been brought to an RDF format. Thus, the basic unit of content here is a triple $t{=}(s, p, o)$, specifying the value o of property p for the resource s. We assume triples are *serialized following the XML syntax for RDF* [6]. As customary in RDF, s, p and o range over the set of all URIs, plus the set of String values in the case of o.

III. Query and View Language

Views and queries are defined in the same language, which can be seen as joins over a specific flavor of tree patterns. VIP2P supports a tree pattern dialect $\mathcal{P}$, a comprehensive presentation of which can be found in [1]. Here we provide a brief overview of $\mathcal{P}$ and then we describe how it was extended to the dialect $\mathcal{P}_{\bowtie}$ for the needs of AnnoVIP.

Each tree pattern node carries a name label, corresponding to an element or attribute name or a word appearing in a text node in some document or annotation (we will use $\underline{\mathtt{w}}$ to denote the word $\mathtt{w}$). Pattern edges correspond either to the child (/) or the descendant (//) relationships between nodes; we assume that a text word is a child of its closest enclosing element or ancestor node. *Due to the special role we attach to URIs, we impose that an URI-labeled view node always appears as a child (not descendant) of another node in the view.* Each node may be decorated with zero or more among the following labels: id, standing for *structural identifier*[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. Finally, each node may be labeled with a predicate of the form $[val{=}c]$ where c is some constant.

We have generalized the tree pattern language to include patterns consisting of more than one tree patterns, joined using some value equality predicates. Formally, let $tp_1, tp_2, \ldots, tp_n \in \mathcal{P}$ be tree patterns. We define a pattern $p \in \mathcal{P}_{\bowtie}$ by combining the above tree patterns using value joins between the values of some elements of the tree patterns, and denote it as the algebraic expression $p = \sigma_{pred}(tp_1 \times tp_2 \times \ldots \times tp_n)$, where $pred$ is the conjunction of all value equality predicates. Thereinafter, we use $p.t_1, p.t_2, \ldots, p.tp_n$ to designate the tree patterns included in the pattern p.

Due to the value joins, more complex views and queries can be defined now. Furthermore, value joins are an important feature for the exploitation of documents *and annotations*. In fact, whenever one wants to retrieve (some part of) a document that is the subject of an annotation, one join is needed; similarly, whenever one wants to "chain" two triples by ensuring that the subject of one is the value of the other, a similar join is needed. These joins are not a feature of one given RDF serialization; rather, they derive from the model built in the very idea of annotation, which is: identify resources by their URI, and have annotations refer to them by their URIs. Value joins are thus naturally present whenever one seeks to jointly exploit resources and related annotations.

Pattern Semantics In [1] we defined an embedding $\phi : tp \rightarrow d$ of the tree pattern $tp \in \mathcal{P}$ in document d as a function associating d nodes to tp nodes, preserving node labels and ancestor-descendant (and parent-child) relationships [9]. For a document set $\mathcal{D}$, the semantics of tp over $\mathcal{D}$, denoted $tp(\mathcal{D})$, are defined as the concatenation of all $tp(d)$, $d \in \mathcal{D}$. Let now p be a pattern belonging to the generalized pattern language $\mathcal{P}_{\bowtie}$ presented above, with $p.tp_1, p.tp_2, \ldots, p.tp_n$ being the tree patterns included in p and $pred$ the conjunction of the value equality predicates. We observe that the embedding of each of these tree patterns may be found in different documents of $\mathcal{D}$. We define the semantics of p over $\mathcal{D}$, denoted $p(\mathcal{D})$, as the algebraic expression $p(\mathcal{D}) = \sigma_{pred}(p.tp_1(\mathcal{D}) \times p.tp_2(\mathcal{D}) \times \ldots \times p.tp_n(\mathcal{D}))$. More intuitively, $p(\mathcal{D})$ represents the join (according to the value equality predicates included in $pred$) of the semantics of $p.tp_1, p.tp_2, \ldots, p.tp_n$ over $\mathcal{D}$.

IV. View Materialization

Assume peer p decides to establish a view $v \in \mathcal{P}_{\bowtie}$. 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) to find a superset S_a of the views that d might affect. Then, p_d

[1] By comparing the structural identifiers of nodes n_1 and n_2, one can decide n_1 is an ancestor of n_2 or not. Many popular examples exist, e.g. [7], [8].

evaluates $v(d)$ for each $v \in S_a$. The case of materializing views consisting of only one tree pattern is presented in [1]. Assume now that a view in S_a contains more than one tree patterns and some value joins between them. It is not possible to materialize such a view only by considering document d, as p_d needs to know if and where, in the whole network, some other content may satisfy a value join with d. A first solution could be to maintain, instead of a tree pattern join view, one view per each tree pattern, and compute view tuples incrementally as new tuples are added to each tree pattern, in the style of incremental maintenance for join views [10]. However, this may lead to accumulating an unbound amount of data, if e.g. many documents matching one view tree pattern are published, which do not join with any other document or annotation.

We will now describe a more efficient technique for the case when join predicates *do not* involve URI attributes. Assume, as above, peer p_d publishes a document d. Let $v' \in S_a$ be a view consisting of two tree patterns, tp_1 and tp_2, with a value join between node n_1 of tp_1 and n_2 of tp_2. If there is an embedding of tp_1 in d, instead of storing the whole tuple in tp_1, we store only the value of n_1 and the URI of d. Assume now a document d' is published and there is an embedding of tp_2 in d'. The publishing peer searches the (value,URI) pairs and finds immediately the documents with which he could join, by comparing his n_2 value with the available n_1 values. Thus, he receives the appropriate documents, joins with them and provides new tuples to the view extent.

Moreover, we have devised a particular technique to treat the materialization of views including joins over URI attributes. This is the case whenever a view queries documents with annotations, as the view will involve joins over virtual URI attributes. Notice that annotations are necessarily published *after* the content they refer to. Thus, when a new document is published, it will not contribute (yet) to join views requiring specific annotations over the document. On the contrary, when a newly published annotation matches a join view, the URI of the annotated element appears in the annotation and the document enclosing this element can thus be identified. The peer that has published the annotation then asks the document peer to compute its corresponding tuples, which are joined with the tuples extracted from the annotation and sent for storage at the site of the view.

V. View-Based Query Rewriting

The rewriting of a query using a set of materialized views is in the crux of the problem we examine. Let q be a $\mathcal{P}_{\bowtie}$ query, and $e(v_1, v_2, \ldots, v_n)$ be an algebraic expression over the $\mathcal{P}_{\bowtie}$ views in $\mathcal{V}$. We say $e(\mathcal{V})$ is an equivalent rewriting of q, if and only if for any database $\mathcal{D}$, $e(v_1(\mathcal{D}), v_2(\mathcal{D}), \ldots, v_k(\mathcal{D})) = q(\mathcal{D})$. Several new algorithms are presented in [1], which are, though, restricted to tree pattern views and queries. Here we provide an extension to these algorithms to support the rewriting of tree pattern join queries by possibly exploiting tree pattern join views.

Algorithm 1: Subset-enum

Input : tree pattern query $tq \in \mathcal{P}$, tree pattern view set TW (views belonging to $\mathcal{P}$)
Output: all minimal canonical rewritings of tq based on TW

```
1 PR ← ∅
2 foreach QTW = {v_1, v_2, ..., v_k} ⊆ TW, |QTW| ≤ |tq| do
3     foreach tuple φ_1, φ_2, ... φ_k of embeddings from
      v_1, v_2, ..., v_k into tq do
4         e ← Views-to-rewriting(TW, φ_1, φ_2, ... φ_k)
5         if e is an equivalent rewriting then
6             add e to PR
7 remove from PR non-minimal rewritings
8 return PR
```

First, we present a slightly different version of the algorithm **Subset-enum** (Algorithm 1), which was initially described in [1]. This algorithm takes as input a tree pattern query $tp \in \mathcal{P}$ and a set of tree pattern views TW (belonging to $\mathcal{P}$), and returns all minimal canonical rewritings of tq based on TW. It iterates over all subsets QTW $\subseteq$ TW whose cardinality is at most equal to the number of nodes of tq ($|\text{QTW}| \leq |tq|$), all embedding combinations from the views into tq, and accumulates rewritings in the set PR. A rewriting r is non-minimal if another rewriting $r' \in PR$ uses a subset of r's views. We notice that in line 4, the algorithm **Views-to-rewriting** is invoked. This algorithm, which is also described in [1], takes a tree pattern query, a set of tree pattern views, as well as their embeddings to the query, and builds a single algebraic expression over all the views, or fails.

We now present the algorithm **Pattern-rewriting** (Algorithm 2), which executes an end-to-end rewriting, given a tree pattern join query $q \in \mathcal{P}_{\bowtie}$ and a set of tree pattern join views $\mathcal{V}$. The algorithm starts by executing a **pruning** step (line 1), during which, any view that will certainly not participate in the rewriting of q is pruned, whereas the rest are added to the set $\mathcal{W}$. In particular, if a view $v \in \mathcal{P}_{\bowtie}$ appears in a rewriting of q, then there exists an embedding $\phi : qt \rightarrow q$ for every tree pattern $qt \in \mathcal{P}$ included in v, such that:

1) ϕ preserves node names
2) if n is a parent of m in v, $\phi(n)$ is an ancestor of $\phi(m)$
3) if m has a value predicate $[val = c_1]$ in q and $\phi(n) = m$, for some v node m, then m must not have a value predicate $[val = c_2]$, if $c_1 \neq c_2$.

Moreover, let m be a node in a tree pattern t_1 and n be a node in a tree pattern t_2. Let ϕ_1 and ϕ_2 be the embeddings of t_1 and t_2 in q, respectively. If t_1 is joined with t_2 through the predicate $m.val = n.val$, then ϕ_1 must be joined with ϕ_2 through the predicate $\phi_1(m).val = \phi_2(n).val$.

View expansion (line 2) then follows. This step is described in [1] in detail, so we don't further analyze it here. Once the view expansion is completed, the algorithm iterates over all

Algorithm 2: Pattern-rewriting

Input : query $q \in \mathcal{P}_{\bowtie}$ consisting of the tree patterns $q.t_1, q.t_2, \ldots, q.t_m \in \mathcal{P}$, view set $\mathcal{V}$ (views belonging to $\mathcal{P}_{\bowtie}$)

Output: all minimal canonical rewritings of q based on $\mathcal{V}$

1 $\mathcal{U} \leftarrow$ prune($\mathcal{V}$, q)
2 $\mathcal{W} \leftarrow \cup_{v \in \mathcal{V}}\ expand(v)$
3 $R \leftarrow \emptyset$
4 **foreach** W $\subseteq \mathcal{W}$, $|\text{W}| \leq |q|$ **do**
5 TW $\leftarrow \cup_{v \in \text{W}}\ (\cup_{tp \in v}\ v.tp)$
6 $PR \leftarrow \emptyset$
7 **foreach** $qt \in \{q.t_1, q.t_2, \ldots, q.t_m\}$ **do**
8 $PR(qt) =$ Subset-enum(qt,TW)
9 **foreach** *combination* $rc = (r_1, r_2, \ldots, r_m)$*, such that* $r_1 \in PR(q.t_1), r_2 \in PR(q.t_2), \ldots, r_m \in PR(q.t_m)$ **do**
10 **if** rc *can result in an equivalent rewriting of* q **then**
11 add rc to R
12 remove from R non-minimal rewritings
13 return R

subsets W $\subseteq \mathcal{W}$, trying to find all possible rewritings using the views in W. To this end, in each iteration, a set TW is built (line 5), containing all the tree patterns (belonging to $\mathcal{P}$) appearing in the views (belonging to $\mathcal{P}_{\bowtie}$) of W. We should remark here that so as to reduce our search space, we used the lemma concerning the bound on the minimal rewriting size, proven in [1], according to which, a minimal canonical rewriting of q uses at most $|q|$ views, where $|q|$ is the number of nodes that q contains.

Then, every tree pattern $qt \in \mathcal{P}$ appearing in q is rewritten based on TW (lines 7-8), using the Algorithm 1, and the equivalent rewritings of every qt are stored in a set $PR(qt)$. In order to build rewritings for the whole query (lines 9-11), we take one rewriting from every set $PR(qt)$, check whether all views included in TW are used in this combination, add all value equality predicates appearing in the views of W, as well as any other projection, value selection or value join needed, and examine whether the emerging expression is an equivalent rewriting of q. The expressions that constitute a rewriting of q, are added to the set R, which is the output of the algorithm.

We should notice that in line 8 of Pattern-rewriting, we use Subset-enum to discover the equivalent rewritings of a tree pattern over a set of tree pattern views. Nevertheless, in [1], we have described three more algorithms for this purpose: Increasing-Subset-Enumeration, Dynamic Programming Rewriting, Depth-First Rewriting. Thus, Subset-enum could be substituted by any of the other three algorithms.

VI. Related works

Our approach relates to many works on XML indexing in DHT networks [11], [12], [13], over which it improves by allowing to declare and exploit complex materialized views to speed-up the processing of specific application queries. In [1], we have provided algorithms which handle efficiently tree pattern views in large networks (1000 peers). This work is the first to jointly consider documents and annotations, and as a consequence, we added value joins to the views and to the queries. Numerous recent works have targeted efficient RDF querying in a centralized setting [14] and based on DHTs [15]. The specificity of our work is to first, combine documents and annotations and second, focus on view maintenance and view-based rewriting.

Within the WebContent project [16], we have devised a DHT-based platform integrating two types of DHT content indices [17]. However, this still did not provide sufficient leeway to establish efficient data access support structures. Moreover, the biggest performance problems we encountered while using the system [17] were due to the frequent joins generated by document-and-annotations queries.

References

[1] I. Manolescu and S. Zoupanos, "Materialized views for P2P XML warehousing," http://vip2p.saclay.inria.fr/papers/paper.pdf, 2009, submitted for publication.
[2] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica, "Towards a common API for structured P2P overlays," in *IPTPS*, 2003.
[3] "Dublic core metadata initiative," http://www.dublincore.org/.
[4] "RDF: Concepts and Abstract Syntax," http://www.w3.org/TR/rdf-concepts/, 2004.
[5] "XML Pointer Language," http://www.w3.org/TR/WD-xptr, 2001.
[6] "RDF/XML Syntax Specification," http://www.w3.org/TR/rdf-syntax-grammar/, 2004.
[7] 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.
[8] 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*, 2002.
[9] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava, "Tree pattern query minimization," *VLDB J.*, vol. 11, no. 4, 2002.
[10] A. Gupta, I. S. Mumick, and V. S. Subrahmanian, "Maintaining views incrementally," in *SIGMOD Conference*, 1993.
[11] S. Abiteboul, I. Manolescu, N. Polyzotis, N. Preda, and C. Sun, "XML processing in DHT networks," in *ICDE*, 2008.
[12] L. Galanis, Y. Wang, S. Jeffery, and D. DeWitt, "Locating data sources in large distributed systems," in *VLDB*, 2003.
[13] P. Rao and B. Moon, "An internet-scale service for publishing and locating XML documents (demo)," in *ICDE*, 2009.
[14] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: sextuple indexing for semantic web data management," *PVLDB*, vol. 1, no. 1, 2008.
[15] E. Liarou, S. Idreos, and M. Koubarakis, "Evaluating conjunctive triple pattern queries over large structured overlay networks," in *ISWC*, 2006.
[16] "The WebContent Semantic Web platform," www.webcontent.fr.
[17] S. Abiteboul, T. Allard, P. Chatalic, G. Gardarin, A. Ghitescu, F. Goasdoué, I. Manolescu, B. Nguyen, M. Ouazara, A. Somani, N. Travers, G. Vasile, and S. Zoupanos, "Webcontent: efficient P2P warehousing of web data (demo)," *PVLDB*, vol. 1, no. 2, 2008.