Structured Materialized Views for XML Queries

Andrei Arion[1] Véronique Benzaken[2] Ioana Manolescu[1]
Yannis Papakonstantinou[3]
[1]INRIA Futurs, France [2]LRI - Univ. Paris-Sud, France [3]CSE Department, UCSD, USA

ABSTRACT

The performance of XML database queries can be greatly enhanced by rewriting them using materialized views. We study the problem of rewriting a query using materialized views, where both the query and the views are described by a tree pattern language, appropriately extended to capture a large XQuery subset. The pattern language features optional nodes and nesting, allowing to capture the data needs of nested XQueries. The language also allows describing storage features such as structural identifiers, which enlarge the space of rewritings. We study pattern containment and equivalent rewriting under the constraints expressed in a structural summary, whose enhanced form also entails integrity constraints. Our approach is implemented in the ULoad [6] prototype and we present a performance analysis.

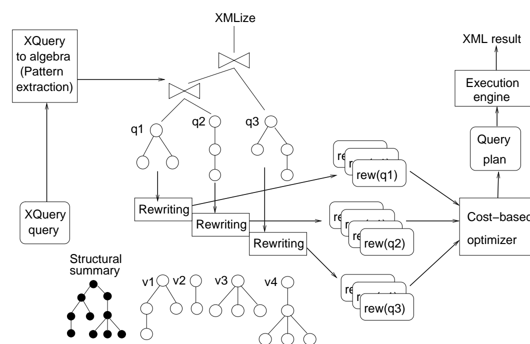


Figure 1: Outline of query processing in ULoad.

1. INTRODUCTION

The structural complexity of XML data and the potentially high costs for processing complex, nested XQuery queries make materialized views an essential tool for XML databases. Indeed, defining a small set of materialized views may result in avoiding complex computations, yielding important performance improvements for a large set of queries. While many works have addressed the topic in the context of the relational model, the issue is a topic of active research in the context of XML. Previous works have mainly focused on XPath [8, 27, 39, 25] and XQuery [15, 12] views. One work [25] considers maximally contained rewriting under constraints.

We study the problem of rewriting a query using materialized views, where both the query and the views are described by extended tree patterns. The rewriting algorithm exploits a set of constraints over the document structure, expressed in a structural summary. We discuss these aspects next.

Tree pattern views. The materialized views we consider are specified using extended tree patterns, including optional and nested edges, value predicates and element identifiers. The tree pattern language has been introduced in [3]. It is not a subset of a query language such as XPath or XQuery (although it includes core XPath [29] and is semantically very close to XQuery, see Section 7). When evaluated on a document, a materialized view yields a collection of (possibly nested) tuples, whose attributes can be simple values, element IDs, XML fragments, or other tuple collections. We consider views with optional and nested edges, as they correspond to the semantics of nested XQuery queries. Tuple-based views are useful because they preserve the relationships between different nodes in a document. For instance, one can store an XML element with its (possibly empty) nested collection of descendants of a given tag and its persistent identifier assigned by the system. Finally, when a view stores persistent IDs for some nodes, our view language allows specifying interesting properties of the IDs, which enlarge the space of possible rewritings (Section 2 will illustrate this). This feature is interesting as special-properties identifiers are increasingly used in research prototypes such as Timber [23] and MonetDB [10] and also in Microsoft SQL Server [31]. Such identifiers enable very efficient XML query processing techniques such as structural joins [1] or the XPath accelerator [22].

Interestingly, our tree pattern language is able to describe XML storage structures advocated in previous works such as [9, 24, 34], as well as many XML index models [14, 33] and custom XPath views [27, 39]. If all storage structures and indices in the system are uniformly described, the rewriting algorithm we present in this paper can jointly use them to rewrite the query.

From XML queries to tree patterns. The queries we consider here are expressed in the same tree pattern formalism used to describe the views. Real user queries, however, are specified in a large subset of XQuery, featuring child and descendant navigation, and nested for-where-return expressions [5]. The global query processing strategy around the rewriting algorithm is outlined in Figure 1. From the user's XQuery query, an algebraic expression is extracted, representing the query as a join expression over one or several tree patterns, perhaps topped with an XMLize operator if new XML elements should be constructed. This pattern extraction process is described in a separate work [5]. Each tree pattern

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '07, September 23-28, 2007, Vienna, Austria.
Copyright 2007 VLDB Endowment, ACM 978-1-59593-649-3/07/09.

is then rewritten separately into a set of alternative algebraic expressions; the rewriting algorithm is the main contribution of the present work. The algebraic rewritings thus obtained are then handled to a cost-based query optimizer, which chooses some combination of rewritings (one for each query pattern), adds the necessary joins operators, may re-order operations etc.

Summary constraints. Practical application domains entail application constraints on the data sources. We consider the constraints encapsulated in a structural summary (or Dataguide [20]), which we then enhance with integrity constraints. Knowledge of constraints increases the space of rewritings, as Section 2 illustrates. There is an interesting interplay between the benefits brought by the special-properties IDs and those of a structural summary. Indeed, while these IDs allow combining views in many ways, the summary helps pruning out useless combinations. The summary benefits come at a modic cost: Dataguides for tree-structured data are typically compact and can be built and maintained in linear time [20]. They can be also used when a schema is unavailable.

Contributions and outline. We focus on the problem of containment and equivalent rewriting of tree pattern queries in the presence of structural and integrity constraints. We consider queries and views expressed in a rich tree pattern formalism, particularly suited for nested XQuery queries, and which extends previously used view [8, 27, 39] and tree pattern [2, 13] formalisms. Given a query and a set of views:

- We characterize the complexity of pattern containment under Dataguides [20] and integrity constraints, and provide a containment decision algorithm, necessary for rewriting.
- We describe a sound and complete view-based rewriting algorithm which produces an algebraic plan combining the tree pattern views. The result of this plan is the same as the query's, for all inputs obeing the Dataguide constraints.
- The containment and rewriting algorithms have been fully implemented in the ULoad prototype, which was recently demonstrated [6]. We report on their practical performance.

The novelty of our work resides in two independent aspects. (i) The expressive power of our view language goes beyond previous XPath-based proposals, and comes close to the needs of XQuery queries, while avoiding structure and identity loss that XQuery-expressed views may bring (see Section 7). We also improve over all previous proposals by exploiting interesting ID properties for rewriting. (ii) Ours is the first work to address XML query rewriting under Dataguide constraints. The Dataguide turns out to be crucial for finding some interesting equivalent rewritings.

This paper is organized as follows. Section 2 presents a motivating example, and Section 3 reviews preliminary definitions. For readability, containment and rewriting algorithms are presented in two steps. Section 4 considers containment and rewriting for a very simple flavor of conjunctive patterns and constraints, while Section 5 extends these results to the full tree pattern language and to richer constraints. Section 6 presents a performance evaluation. We review related works, and conclude.

2. MOTIVATING EXAMPLE

As an example illustrating key concepts, requirements and contributions, consider the following XQuery:

```
for $x in document("XMark.xml")//item[//mail] return
⟨res⟩ {$x/name/text(),
        for $y in $x//listitem return
        ⟨key⟩ {$y//keyword} ⟨/key⟩} ⟨/res⟩
```

Figure 2(a) shows a simplified XMark document fragment. At the right of each node's label, we show the node's identifier, e.g. n_1, n_2 etc.

We exploit XML structural summaries to increase the rewriting opportunities. In short, a structural summary (or strong Dataguide [20]) of an XML document is a tree, including all paths occurring in the document. Figure 2(b) shows the structural summary of the document in Figure 2(a).

Each view is defined by an extended tree pattern and, evaluated on a document, produces a nested table which may include null values. Figure 2(c) depicts the definitions of views V_1 and V_2, and the result obtained by evaluating the views over the sample document above. As is common in tree pattern languages, / denotes child and // denotes descendant relationships. Variables, such as ID, C, and V label certain nodes of the tree pattern. Dashed edges indicate that a tuple should be produced even if the (sub)tree pattern hanging at the dashed edge cannot bind to a corresponding subtree of the input. For example, consider the last tuple of V_1: The variable ID is bound to n_{21}, despite the fact that n_{21} has no ⟨bold⟩ descendant; V is bound to null ($\perp$).

A pattern edge may be labeled n. In this case, there will be a single attribute in the tuple for the subtree pattern hanging below the n-edge. The content of this attribute is a relation whose tuples are the bindings of the variables of the subtree. For example, the A attribute of V_1 corresponds to the subtree under the single n-edge of the tree pattern. Its values are relations of unary tuples, whose only attribute is the variable C of the pattern hanging at the n-edge.V_1 stores, for every //regions//* element, four information items: (1) Its identifier ID. We consider identifiers are simple atomic values. (2) The grouped set of content of its possible parlist/listitem grandchildren nodes. The content C of a node denotes the subtree rooted in the node, which the view may store directly (perhaps in a compact encoding), or as a pointer to the subtree stored elsewhere. In all cases, downward navigation is possible inside a C attribute. (3) The value (text children) of its possible bold descendants. View V_2 stores, for every //regions//item element, its identifier ID, and value V.

Rewriting can benefit from knowledge of the structure of the document and of the structure IDs. We describe our contributions in the area using cases from the running example.

Summary-based rewriting Consider the following rewriting opportunities that are enabled by the structural summary. First, although the tree pattern of V_1 does not explicitly indicate that V_1 stores data from ⟨item⟩ nodes, V_1 is useful if the structural summary in Figure 2(b) guarantees that all children of ⟨region⟩ that have ⟨description⟩ children are labeled item.

Second, in the absence of structural summaries, evaluation of the $y//keyword path of the query is impossible since neither V_1 nor V_2 store data from keyword nodes. However, if the structural summary implies that all //region//item//keyword nodes are descendants of some //region//item/description/parlist/listitem, we can extract the keyword elements by navigating inside the content of ⟨listitem⟩ nodes, stored in the $A.C$ attribute of V_1.

Third, V_1 stores //region//*/description/parlist/listitem elements, while the query requires all ⟨listitem⟩ descendants of //regions//item. V_1's data is sufficient for the query, if the summary ensures that //regions//item//listitem and //regions//*/description/parlist/listitem deliver the same data.

Summary-based optimization The rewritten query can be more efficient if it utilizes the knowledge of the structural summary. For example, V_1 may store some tuples that should not contribute to the query, namely from ⟨item⟩ nodes lacking ⟨mail⟩ descendants. In this case, using V_1 to rewrite our sample query requires checking

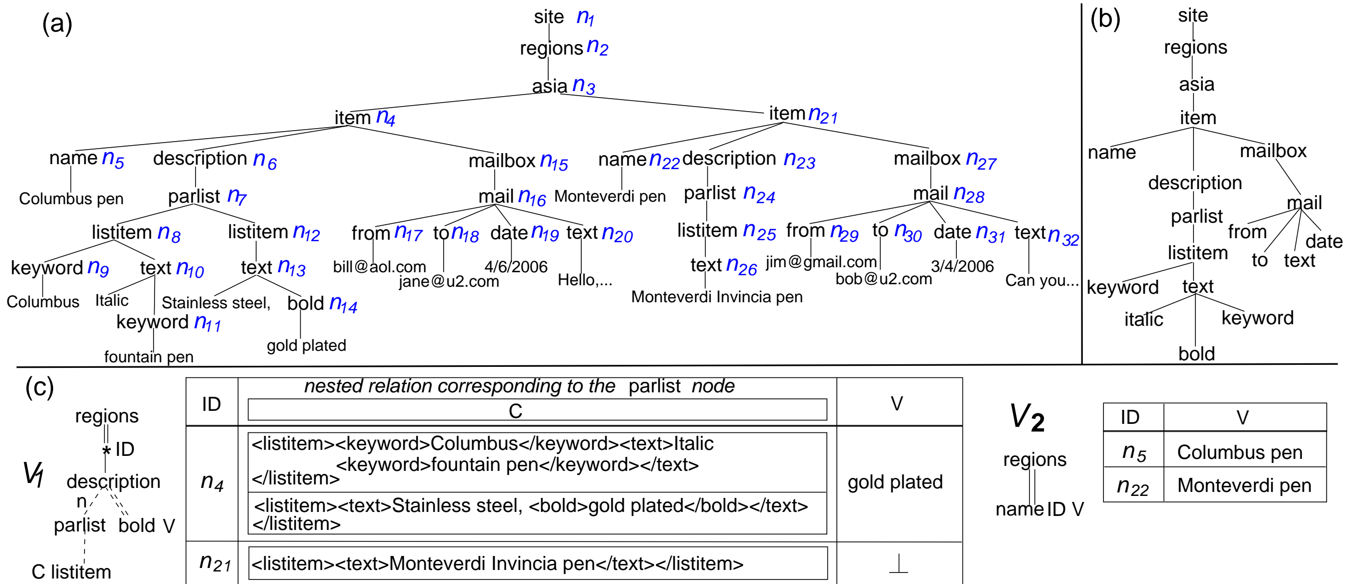


Figure 2: (a) XMark document fragment, (b) its structural summary and (c) two materialized views.

for the presence of ⟨mail⟩ descendants in the C attribute of each V_1 tuple. If all ⟨item⟩ nodes have ⟨mail⟩ descendants, V_1 only stores useful data, and can be used directly.

The above requires using structural information about the document and/or integrity constraints, which may come from a DTD or XML Schema, or from other structural XML summaries, such as Dataguides [20]. The XMark DTD [38] can be used for such reasoning, however, it does not allow deciding that //regions//item//listitem and //regions//*/description/parlist/listitem bind to the same data. The reason is that ⟨parlist⟩ and ⟨listitem⟩ elements are recursive in the DTD, and recursion depth is unbound by DTDs or XML Schemas. While recursion is frequent in XML, it rarely unfolds at important depths [28]. A Dataguide is more precise, as it only accounts for the paths occurring in the data; it also offers some protection against a lax DTD which "hides" interesting data regularity properties.

Rewriting with rich patterns In addition to structural summaries, we also make use of the rich features of the tree patterns, such as nesting and optionality. For example, in V_1, ⟨listitem⟩ elements are optional, that is, V_1 (also) stores data from ⟨item⟩ elements without ⟨listitem⟩ descendants. This fits well the query, which must indeed produce output even for such ⟨item⟩ elements. The nesting of ⟨listitem⟩ elements under their ⟨item⟩ ancestor is also favorable to the query, which must output such ⟨listitem⟩ nodes grouped in a single ⟨res⟩ node. Thus, the single view V_1 may be used to rewrite *across nested FLWR blocks*.

Exploiting ID properties Maintaining structural IDs enables opportunities for reassembling fragments of the input as needed. For example, data from ⟨name⟩ nodes can only be found in V_2. V_1 and V_2 have no common node, so they cannot be simply joined. If, however, the identifiers stored in the views carry information on their structural relationships, combining V_1 and V_2 may be possible. For instance, *structural IDs* allow deciding whether an element is a parent (ancestor) of another by comparing their IDs. Many popular ID schemes have this property [1, 31, 35]. Assuming structural IDs are used, V_1 and V_2 can be combined by a *structural join* [1] on their attributes $V_1.ID$ and $V_2.ID$. Furthermore, some ID schemes also allow inferring an element's ID from the ID of one of its children [31, 35]. Assuming V_1 stored the ID of ⟨parlist⟩ nodes, we could derive from it the ID of their parent ⟨description⟩ nodes, and

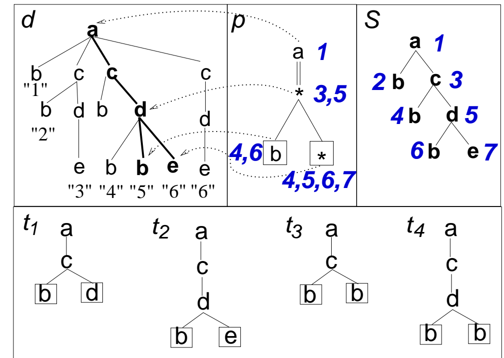


Figure 3: Sample document d, its summary S, pattern p, and canonical model $mod_S(p) = \{t_1, t_2, t_3, t_4\}$.

use it in other rewritings. Realizing the rewriting opportunities requires ID property information attached to the views, and reasoning on these properties during query rewriting. Observe that V_1 and V_2, together, contain all the data needed to build the query result *only if* the stored IDs are structural.

3. PRELIMINARIES

Data model We view an XML document as an unranked labeled ordered tree. Every node n has (i) a unique identity from a set $\mathcal{I}$, (ii) a tag $label(n)$ from a set $\mathcal{L}$, which corresponds to the element or attribute name, and (iii) may have a value from a set $\mathcal{A}$, which corresponds to atomic values of the document. We may denote trees in a simple parenthesized notation based on node labels and ignoring node IDs, e.g. $a(b\ c(d))$. Figure 3 depicts a sample document d, where node values appear under node labels, e.g. "1", "2" etc.

We denote that node n_1 is node n_2's parent as $n_1 \prec n_2$ and the fact that n_1 is an ancestor of n_2 as $n_1 \prec\!\!\prec n_2$.

Conjunctive tree patterns We recall the classical notions of conjunctive tree patterns and embeddings [2, 29]. A conjunctive tree

pattern p is a tree, whose nodes are labeled from members of $\mathcal{L} \cup \{*\}$, and whose edges are labeled / or //. A distinguished subset of p nodes are called *return nodes of* p. Figure 3 shows a pattern p, whose return nodes are enclosed in boxes.

An *embedding* of a conjunctive tree pattern p into a document d is a function $e : nodes(p) \to nodes(d)$ such that:

- For any $n \in nodes(p)$, if $label(n) \neq *$, then $label(e(n)) = label(n)$.
- e maps the root of p into the root of d.
- For any $n_1, n_2 \in nodes(p)$ such that n_2 is a /-child of n_1, $e(n_2)$ is a child of $e(n_1)$.
- For any $n_1, n_2 \in nodes(p)$ such that n_2 is a //-child of n_1, $e(n_2)$ is a descendant of $e(n_1)$.

Dotted arrows in Figure 3 illustrate an embedding.

The result of evaluating a conjunctive tree pattern p, whose return nodes are $n_1^p, \ldots, n_k^p$, on an XML document d, is the set $p(d)$ consisting of all tuples $(n_1^d, \ldots, n_k^d)$ where $n_1^d, \ldots, n_k^d$ are document nodes and there exists an embedding e of p in d such that $e(n_i^p) = n_i^d, i = 1, \ldots, k$.

Given a pattern p, a tree t and an embedding $e : p \to t$, we denote by $e(p)$ the tree that consists of the nodes $e(n)$ to which the nodes of p map to, and the edges that connect such nodes. For example, in Figure 3, $e(p)$ is shown in bold. We may use the notation $u \in e(p)$ to denote that node u appears in the tree $e(p)$. In Figure 3, $e(p)$ has more nodes than p, since the intermediary c node also belongs to $e(p)$.

Notation: path Given a document d, a *path* is a succession of /-separated labels $/l_1/l_2/\ldots/l_k$, $k \leq 1$, such that l_1 is the label of d's root, l_2 is the label of one of the root's children etc. Only node labels (not values) appear in paths. We say a node n is *on path* p if the label path going down from the root to node n is p.

Summaries The *summary* of d, denoted $S(d)$, is a tree, such that there is a label and parent-preserving mapping $\phi : d \to S(d)$, mapping all d nodes reachable by the same path p from d's root to the same node $n_p \in S(d)$. Summaries correspond to strong Dataguides [20] of tree-structured data. In Figure 3, S is the summary of document d.

A document d *conforms to* a summary S_1, denoted $S_1 \models d$, *iff* $S(d) = S_1$.

Notations: paths and summary nodes Given a summary S, the set of S nodes is clearly in bijection with the set of S paths (which is the set of paths in any document conforming to S). For ease of explanation, we may refer to a path by its corresponding summary node, or vice-versa.

3.1 Summary-based canonical model

Let p be a conjunctive tree pattern, and S be a summary. Let $e : p \to S$ be an embedding of p in S. The *canonical tree derived from* e, denoted t_e, is obtained as follows:

- For each $n \in p$, t_e contains a distinguished node whose label is that of $e(n)$. When n is a returning node in p, we say $e(n)$ is a returning node in t_e.
- Let $n \in p$ be a node and $m_1, m_2, \ldots, m_k$ its children. Then, the t_e node corresponding to $e(n)$ has exactly k children, and for $1 \leq i \leq k$, its i-th child consists of a parent-child chain of nodes, whose labels are those connecting $e(n)$ to $e(m_i)$ in S.

For instance, in Figure 3, an embedding $e_1 : p \to S$ maps the upper $*$ in p to the S node numbered 3, and the lower returning $*$ node in p to the S node numbered 5. The tree t_1 in Figure 3 is the canonical tree derived from e_1. Similarly, another embedding $e_2 : p \to S$ associates the upper $*$ node in p to the S node numbered 5, and the lower $*$ node to the S tree numbered 7. The tree t_2 in Figure 3 is the canonical tree derived from e_2. Note that an embedding needs not to be an injective function. The return nodes of p can be embedded into the same b node in S, yielding the canonical trees t_3 and t_4.

Let the return nodes in p be $n_1^p, \ldots, n_k^p$. Then for every tree $t_e \in mod_S(p)$ corresponding to an embedding e, the tuple $(e(n_1^p), \ldots, e(n_k^p))$ is called *the return tuple of* t_e. Note that two different trees $t_1, t_2 \in mod_S(p)$ may have the same return tuples.

The S*-canonical model of* p, denoted $mod_S(p)$, is the set of the canonical trees obtained from all possible embeddings of p in S. Clearly, for any canonical tree t_e, $S \models t_e$.

Observe that two distinct embeddings may yield the same canonical tree. For instance, let p' be the pattern $/a//*//e$ where b is the returning node, and consider the following two embeddings of p in the summary S in Figure 3:

- e'_1 maps the $*$ node of p' to the S node numbered 3;
- e'_2 maps the $*$ node of p' to the S node numbered 5.

The canonical trees derived from e'_1 and e'_2 coincide. When defining S, we consider it duplicate-free.

In Figure 3, for the represented pattern p and summary S, we have $mod_S(p) = \{t_1, t_2, t_3, t_4\}$.

In the following, we use the term *subtree* in the following sense. We say a tree t' is a subtree of the tree t if (i) t' and t have the same root, (ii) the nodes of t' are a subset of the nodes of t and (iii) the edges of t' are a subset of the edges of t.

PROPOSITION 3.1. Let t be a tree and S be a summary such that $S \models t$, p be a k-ary conjunctive pattern, and $\{n_1^t, \ldots, n_k^t\} \subseteq nodes(t)$.
$(n_1^t, \ldots, n_k^t) \in p(t) \Leftrightarrow \exists\, t_e \in mod_S(p)$ such that:

1. t has a subtree isomorphic to t_e. For simplicity, we shall simply say t_e is a subtree of t.
2. For every $0 \leq i \leq k$, node n_i^t is on path n_i^S, where n_i^S is the i-th return node of t_e.

$\lhd$

Proof:
$\Leftarrow$: Let $e : p \to S$ be one of the embeddings associated by to t_e (recall that several such embeddings may exist). We define $e' : p \to t$ as follows: for every $n \in p$, $e'(n) = e(n)$, which is safe since $e(p) \subseteq nodes(t_e) \subseteq nodes(t)$. Clearly, e' is an embedding, and $e(n_i^p) = n_i^t$ for every $0 \leq i \leq k$, thus $(n_1^t, \ldots, n_k^t) \in p(t)$.
$\Rightarrow$: By definition, if $(n_1^t, \ldots, n_k^t) \in p(t)$, there exists an embedding $e : p \to t$, such that $e(n_i^p) = n_i^t$ for every $0 \leq i \leq k$. We denote by $e_S : p \to S$ the embedding obtained from e, by setting $e_S(n)$ to be the path of $e(n)$ for each node n of p. Let t_e be the $mod_S(p)$ tree corresponding to e_S. We show that t_e is a subtree of t.

Let n be a t_e node such that $n = e_S(n_p)$ for some $n_p \in p$. Then, n is the path of $e(n_p)$, and since e is an embedding of p in t, then n belongs to t. Thus, all the images of p nodes through e_S belong to t.

Now consider a t_e node n, and let us prove that its children also belong to t. Let m be a direct child of n. Then, by definition of t_e,

m participates in a chain of nodes connecting $e_S(n_p)$ to $e_S(m_p)$, for some m_p child of n_p in p. By definition of e_S, $e_S(m_p)$ is the path of $e(m_p) \in t$, thus the chain of nodes between $e(n_p)$ and $e(m_p)$ belongs to t, thus all edges and nodes between these two nodes (including m) belong to t. Thus, t_e is a subtree of t.

To see that for each i, n_i^d is on path n_i^e, observe that n_i^d is $e(n_i^p)$ for some returning node n_i^p of p, and furthermore $e_S(n_i^p)$ is the path of n_i^d and is also n_i^e.

For example, in Figure 3, bold lines and node names trace a d subtree isomorphic to $t_2 \in mod_S(p)$ (recall t_2 from Figure 3). For the sample document and pattern, the thick-lined subtree is the one Proposition 3.1 requires in order for the boxed nodes in d to belong to $p(d)$.

A pattern p is said *S-unsatisfiable* if for any document d such that $S \models d$, $p(d) = \emptyset$. The above proposition provides a convenient means to test satisfiability: p is S-satisfiable iff $mod_S(p) \neq \emptyset$.

DEFINITION 3.1. Let S be a summary, p be a pattern, and n a node in p. The set of *paths associated to* n consists of those S nodes s_n, such that for some embedding $e : p \to S$, $e(n) = s_n$. $\triangleleft$

At right in Figure 3, the pattern p is repeated, showing next to each node (in italic font) the paths associated to that node.

The paths associated to all p nodes can be computed in $O(|p| \times |S|)$ time and space complexity [21].

4. SUMMARY-BASED CONTAINMENT AND REWRITING OF CONJUNCTIVE PATTERNS

4.1 Summary-based containment

We start by defining pattern containment under summary constraints:

DEFINITION 4.1. Let p, p' be two tree patterns, and S be a summary. We say p *is S-contained in* p', denoted $p \subseteq_S p'$, *iff* for any t such that $S \models t$, $p(t) \subseteq p'(t)$. $\triangleleft$

A practical method for deciding containment is stated in the following proposition:

PROPOSITION 4.1. Let p, p' be two conjunctive k-ary tree patterns and S a summary. The following are equivalent:

1. $p \subseteq_S p'$
2. $\forall\ t_p \in mod_S(p)\ \exists\ t_{p'} \in mod_S(p')$ such that (i) $t_{p'}$ is a subtree of t_p and (ii) $t_p, t_{p'}$ have the same return nodes.
3. $\forall\ t_p \in mod_S(p)$ whose return nodes are $(n_1^t, \ldots, n_k^t)$, we have $(n_1^t, \ldots, n_k^t) \in p'(t_p)$. $\triangleleft$

Proof:
In order to prove the equivalences, note that by definition, $p \subseteq_S p'$ is equivalent to: $\forall\ t$ such that $S \models t$, and nodes $n_1^t, \ldots, n_k^t$ of t:

$$(n_1^t, \ldots, n_k^t) \in p(t) \Rightarrow (n_1^t, \ldots, n_k^t) \in p'(t).$$

For any such t and $n_1^t, \ldots, n_k^t$, let $n_1^S, \ldots, n_k^S$ be the S nodes corresponding to the paths of $n_1^t, \ldots$, respectively n_k^t in t. Then, $p \subseteq_S p'$ is equivalent to:

$(*)$ $\forall\ t$ such that $S \models t$, $\{n_1^t, \ldots, n_k^t\}$ nodes of t, $S_1 \Rightarrow S_2$

where S_1 is:

$$\exists\ t_e \in mod_S(p) \text{ such that } t_e \text{ is a subtree of } t \text{ and } (n_1^S, \ldots, n_k^S) \text{ are the return nodes of } t_e$$

and S_2 is:

$$\exists\ t_{e'} \in mod_S(p') \text{ such that } t_{e'} \text{ is a subtree of } t \text{ and } (n_1^S, \ldots, n_k^S) \text{ are the return nodes of } t'_e$$

(1) $\Rightarrow$ (2): if $p \subseteq_S p'$, let the role of t in $(*)$ be successively played by all $t_e \in mod_S(p)$ (clearly, $S \models t_e$). Each such t_e naturally contains a subtree (namely, itself) satisfying S_1 above, and since $S_1 \Rightarrow S_2$, t_e must also contain a subtree $t_{e'} \in mod_S(p')$ with the same return nodes as t_e.

(2) $\Rightarrow$ (1): let t be a tree and $(n_1^t, \ldots, n_k^t) \in p(t)$. By Proposition 3.1, t contains a subtree $t_e \in mod_S(p)$, such that the return nodes of t are those of t_e, namely $(n_1^t, \ldots, n_k^t)$. By (2), t_e contains a subtree $t_{e'} \in mod_S(p')$ with the same return nodes, and $t_{e'}$ is a subtree of t, thus (again by Proposition 3.1) $(n_1^t, \ldots, n_k^t) \in p'(t)$.

(2) $\Leftrightarrow$ (3) follows directly from Proposition 3.1.

Proposition 4.1 gives an algorithm for testing $p \subseteq_S p'$: compute $mod_S(p)$, then test that $(n_1^S, \ldots, n_k^S) \in p'(t_e)$ for every $t_e \in mod_S(p)$, where $(n_1^S, \ldots, n_k^S)$ are the return nodes of p. The complexity of this algorithm is $O(|mod_S(p)| \times |S| \times |p| \times |p'|)$, since each $mod_S(p)$ tree has at most $|S| \times |p|$ nodes., and $p'(t_e)$ can be computed in $|t_e| \times |p'|$ [21]. In the worst case, $|mod_S(p)|$ is $|S|^{|p|}$. This occurs when any p node matches any S node, e.g. if all p nodes are labeled $*$, and p consists of only the root and // children. For practical queries, however, $|mod_S(p)|$ is much smaller, as Section 6 shows.

A simple extension of Proposition 4.1 addresses containment for unions of patterns:

PROPOSITION 4.2. Let $p, p'_1, \ldots, p'_m$ be k-ary conjunctive patterns and S be a summary. Then, $p \subseteq_S (p'_1 \cup \ldots \cup p'_m) \Leftrightarrow$ for every $t_e \in mod_S(p)$ such that $(n_1, \ldots, n_k)$ are the return nodes of t_e, there exists some $1 \leq i \leq m$ such that $(n_1, \ldots, n_k) \in p'_i(t_e)$. $\triangleleft$

We define *S-equivalence* as two-way containment, and denote it $\equiv_S$. When S is known, we simply call it equivalence.

4.2 Summary-based rewriting

Let $p_1, \ldots, p_n$ and q be some patterns and S be a summary. The rewritings we consider are logical algebraic expressions (or simply plans) built with the patterns p_i, and with the following operators: $\cup$ (duplicate-preserving union), $\bowtie_=$ (join pairing input tuples which contain exactly the same node), $\bowtie_\prec$ and $\bowtie_{\prec\prec}$ (structural joins, pairing input tuple whenever nodes from the left input are parents/ancestors of nodes from the right input), and π (duplicate-preserving projection)[1].

Observe that allowing unions in rewritings leads to finding some rewritings for cases where no rewriting could be found without them. For instance, in Figure 5, the only rewriting of p_1 based on q and p_3 is $q \cup p_3$. This contrasts with traditional conjunctive query rewriting based on conjunctive views [26], and is due to the summary constraints.

A set of plans is said *redundant* if it contains two plans e_1, e_2 returning the same data on any XML document (thus, regardless of any summary constraints). For instance, for any pattern p, if

[1]Other operators (σ, nesting and unnesting, and XPath navigation) will be introduced for more complex patterns in Section 5.6.

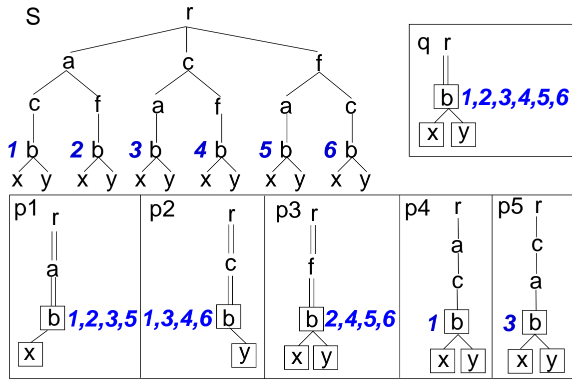


Figure 4: Summary S, query q and patterns.

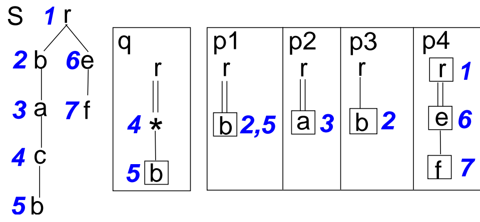


Figure 5: Pattern join configuration.

$e_1 = \pi_{n_1}(p)$ and $e_2 = \pi_{n_1}(\pi_{n_1,n_2}(p))$, the set $\{e_1, e_2\}$ is clearly redundant. We are not interested in redundant plans, since our focus is on rewriting under summary constraints. Furthermore, e_1 is typically preferrable to e_2 in the example above, since e_1 is more concise. More generally, among all plans e that are S-equivalent to q, and also equivalent among themselves independently of S, we are interested in finding a *minimal* plan, i.e., one having the smallest number of operators (there may be several minimal plans, which we regard as equally interesting).

Our rewriting problem can thus be formally stated as: find a maximal, non-redundant set of plans e over $p_1, \ldots, p_n$, such that each plan e in the set satisfies $e \equiv_S q$.

Our query rewriting follows the general "generate-and-test" approach: produce candidate rewritings, and test their equivalence to the query. We first consider testing.

We need to test the S-equivalence between a plan e and a query pattern q. The usage of different formalisms for e and q has its advantages: tree patterns are suited for queries, while an algebraic rewriting language is easy to translate in executable plans. However, testing S-equivalence is more natural on patterns. Therefore, for each algebraic rewriting e, the rewriting algorithm builds an S-equivalent pattern p_e, and it is p_e that will be tested for equivalence to q. However, all plans do not have an S-equivalent pattern ! For example, in Figure 4, no pattern is S-equivalent to $p_1 \bowtie_{b=b} p_2$. The intuition is that we can't decide whether a should be an ancestor, or a descendant of c in the hypothetic equivalent pattern. Fortunately, it can be shown [4] that any plan is S-equivalent to some *union* of patterns. For example, $p_1 \bowtie_{b=b} p_2 \equiv_S (p_4 \cup p_5)$ in Figure 4. Thus, to test whether $e \equiv_S q$, we can rely on our algorithms for testing the S-equivalence of q with (a union of) patterns corresponding to e. The containment tests may be expensive, thus the importance of a rewriting generation strategy that does not produce many unsuccessful ones.

Let us now consider possible generation strategies. Conjunctive query rewriting, in the relational setting [26] as well as in more recent XML-oriented incarnations [15], follows a "bucket" approach, collecting all possible rewritings for every query atom (or node), and combining such partial rewritings into increasingly larger candidate rewritings.

Algorithm 1: Summary-based pattern rewriting

Input : summary S, patterns $p_1, \ldots, p_n, q$
Output: rewritings of q using $p_1, \ldots, p_n$

1. $M_0 \leftarrow \{(p_i, p_i) \mid 1 \leq i \leq n\}$; $M \leftarrow M_0$
2. **repeat**
3.   **foreach** $(e_i, p_i) \in M, (e_j, p_j) \in M_0$ **do**
4.   **foreach** *possible way of joining* e_i *and* e_j *using* $\bowtie_{=_{id}}, \bowtie_{\prec}, \bowtie_{\prec\prec}$ **do**
5. $(e, p) \leftarrow (e_i, p_i) \bowtie (e_j, p_j)$
6.   **if** $p \neq p_i$ *and* $p \neq p_j$ **then**
7.   **if** $p \equiv_S q$ **then**
8.   output l
9.   **else**
10.   **if** $|e| \leq |q| \times |S|$ **then**
11. $M \leftarrow M \cup \{(e, p)\}$
12. **until** M *is stationary*
13. **foreach** *minimal* $N \subseteq M$ *s.t.* $\cup_{(e,p) \in N}\, p \equiv_S q$ **do**
14.   output $\cup_{(e,p) \in N}\, p$

Bucket-style generation of rewriting is not adapted in the presence of summary constraints. First, rewriting must consider also views that do not fit in any bucket, i.e. do not cover any query atom. For instance, in Figure 5, q asks for b elements at least two levels below the root, while p_1 provides all b elements, including some not in q. The pattern p_2 does not cover any q nodes, yet $(p_2 \bowtie_{a \prec\prec b} p_1) \equiv_S q$. Second, a rewriting may fail to cover some query nodes, yet be equivalent to the query. For instance, consider a summary $S = r(a(b))$, the query $q = /r//a//b$, and the pattern $p_1 = /r//b$. Clearly, $p_1 \equiv_S q$, yet p_1 lacks an a node (implicitly present above b, due to the S constraints).

As a consequence, our basic generation approach should not start with buckets, but proceed in an inflationary manner, combining views into increasingly larger plans.

Summary impact on the search space When should rewriting generation stop ? Since our patterns include a limited form of recursion (descendant edges), it may seem that the search space is infinite; consider rewriting a $//$ query based on a two-node parent-child view. Indeed, under DTD constraints, the size of a join rewriting in this case is unbound. Fortunately, in our setting, a summary limits the maximal depth of a parent-child chain. More generally, given a query q and summary S, the number of views used in a join plan e, part of a minimal rewriting of q, is at most $|q| \times |S|$, where $|q|$ is the number of q nodes [40].

Summary-based pruning A summary enables restricting the set of views initially considered for rewriting, without losing solutions. Assume that for a view p_i, for any $n_p \in nodes(p_i) \setminus root(p_i)$ and x associated path of n_p, and for any $n_q \in nodes(q) \setminus root(q)$ and y associated path of n_q, $x \neq y$, x is neither an ancestor nor a descendant of y. Then, the rewriting algorithm can ignore p_i. The intuition is p_i's data belongs to different parts of the document than those needed by the query. An example is pattern p_4 for the rewriting of q in Figure 5.

The summary can also be used to restrict the set of intermediary rewritings. The rewriting algorithm manipulates (e, p) (plan, pattern) pairs, where $e \equiv_S p$. Consider two pairs (e_x, p_x) and

(e_y, p_y), and a possible join result $(e_z, p_z) = (e_x, p_x) \bowtie (e_y, p_y)$. (i) If p_z is S-unsatisfiable, we can discard (e_z, p_z). (ii) If the produced pattern (or pattern union) p_z coincides with p_x, we may discard the partial rewriting (e_z, p_z), since any complete rewriting e' based on e_z is non-minimal (e_z can be replaced with e_x, which is smaller). The role of S here is to prune non-minimal plans.

Summary-based reduction of containment tests Structural summaries also allow reducing the number of containment tests performed during rewriting. Since containment is only defined on same-arity patterns, prior to testing whether $p \subseteq_S q$, one must identify k return nodes of p, where k is the arity of q, extract from p a pattern p' returning those k nodes, then test if $p' \subseteq_S q$. If p's arity is smaller than k, clearly $p \not\equiv_S q$. Otherwise, there are many ways of choosing k return nodes of p, which may lead to a large number of containment tests. However, if $p \subseteq_S q$, then for every return node n_i of p and corresponding return node m_i of q, the S paths associated to n_i must be a subset of the S paths associated to m_i. We only test containment for those choices of k nodes of p satisfying this path condition.

Rewriting algorithm Algorithm 1 outlines the rewriting process discussed above. M_0 is the set of initial (plan, pattern) pairs, and M is the set of intermediary rewritings. Initial view pruning is applied prior to step 1, while partial rewritings are pruned in step 6 (for conciseness, some pruning steps are omitted). Lines 13-14 compute union plans. The algorithm's soudness is guaranteed by the equivalence test (line 7). The algorithm is complete due to its exhaustive search. The search space is finite thanks to the summary: it can be shown that all patterns above a certain size (thus their equivalent plans) have an equivalent smaller pattern (thus plan). The complexity is determined by the size of the search space, multiplied by the complexity of an equivalence test. The search space size is in $O(2^{C_{|p|}^{|q|}})$, where $|p| = \Sigma_{i=1,\ldots,n}|nodes(p_i)|$ and $|q| = |nodes(q)|$ (the formula assumes that every p_i node, $1 \leq i \leq n$, can be used to rewrite every q node).

5. COMPLEX SUMMARIES AND PATTERNS

We now present a set of useful, mutually orthogonal extensions to the tree pattern containment and rewriting problems discussed previously. The extensions consist of using more complex summaries, enriched with a class of integrity constraints (Section 5.1), respectively, more complex patterns. Section 5.2 considers patterns endowed with value predicates, Section 5.3 addresses patterns with optional edges, Section 5.4 describes containment of patterns which may store several data items for a given node, and Section 5.5 enriches patterns with nested edges. Finally, Section 5.6 outlines the impact of these extensions on the rewriting algorithm.

5.1 Enhanced summaries

Useful rewriting information may be derived from an *enhanced summary*, or summary with integrity constraints. An enhanced summary S of document d is obtained from its simple summary S_0 by distinguishing a set of *strong* edges. Let n_1 be an S node, and n_2 be a child of n_1. The edge between n_1 and n_2 is *strong* if every d node on path n_1 has at least one child on path n_2. Such edges reflect the presence of integrity constraints, obtained from a DTD or XML Schema, or by counting nodes when building the summary. We depict strong edges by thick lines, as in Figure 6.

A document d conforms to an enhanced summary S iff d conforms to S viewed as a simple summary, and it respects the parent-child constraints enforced by strong S edges.

Enhanced summaries modify the definition of canonical models. The canonical model of p based on the enhanced summary S,

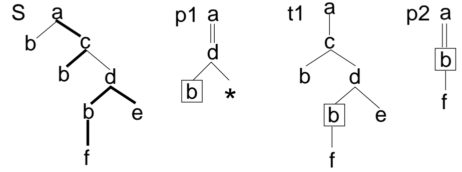


Figure 6: Enhanced summary example.

denoted $mod_S(p)$, is obtained as follows. For every embedding $e : p \to S$, $mod_S(p)$ includes the minimal tree t_e containing: (i) all nodes in $e(p)$ and (ii) all nodes connected to some node in $e(p)$ by a chain of strong edges only. For example, in Figure 6, the canonical model of pattern p_1 consists of the tree t_1, where the b child of the c node and the f node appear due to the strong edges above them in S. Enhanced summary-based containment is decided similarly to the simple summary case. For example, applying Proposition 4.1 in Figure 6 yields $p_1 \equiv_S p_2$.

Besides strong summaries, we consider another class of integrity constraints. Assume a distinguished subset of S edges are *one-to-one*, meaning every XML node on the parent path s_1 has exactly one child node on the child path s_2. Canonical pattern models can be easily adapted to the presence of one-to-one edges.

5.2 Value predicates on pattern nodes

A useful feature consists of attaching *value predicates* to pattern nodes. Summary-based containment in this case requires some modifications, as follows.

A *decorated conjunctive pattern* is a conjunctive pattern where each node n is annotated with a logical formula $\phi_n(v)$, where the free variable v represents the node's value. The formula $\phi_n(v)$ is either T (true), F (false), or an expression composed of atoms of the form $v\,\theta\,c$, where $\theta \in \{=, <, >\}$, c is some $\mathcal{A}$ constant, using $\vee$ and $\wedge$. In Figure 7, $p_{\phi 1} - p_{\phi 4}$ are decorated patterns. Next to their return nodes we show the corresponding path annotations, based on the summary in Figure 3.

We assume $\mathcal{A}$, the domain of atomic values, is totally ordered and enumerable (corresponding to machine-representable atomic values). Then, any $\phi(v)$ can be represented compactly (e.g. by a union of disjoint intervals of $\mathcal{A}$ on which $\phi(v)$ holds), and for any formulas $\phi_1(v), \phi_2(v)$, one can easily compute $\neg\phi_1(v)$, $\phi_1 \vee \phi_2$, $\phi_1 \wedge \phi_2$, and $\phi_1(v) \Rightarrow \phi_2(v)$.

We extend our model of labeled trees to *decorated labeled trees*, whereas instead of an $\mathcal{A}$ value, every node n is decorated with a (non-F) formula $\phi_n(v)$ as described above. Observe that simple labeled trees are particular cases of decorated ones, where for every n, $\phi_n(v)$ is $v = v_n$, where $v_n \in \mathcal{A}$ is n's value.

A *decorated embedding* of a decorated pattern p_ϕ into a decorated tree t_ϕ is an embedding e, such that for any $n \in nodes(p_\phi)$, $\phi_{e(n)}(v) \Rightarrow \phi_n(v)$. Figure 7 illustrates a decorated embedding from $p_{\phi 1}$ to t. The semantics of a decorated pattern is defined similarly to the simple ones, based on decorated embeddings.

Given a summary S, the S canonical model $mod_S(p_\phi)$ of a decorated pattern p_ϕ, is obtained from $mod_S(p)$ (where p is the pattern obtained by erasing p_ϕ's formulas) by decorating, in every tree $t_e \in mod_S(p)$ corresponding to an embedding e: (i) each node $s = e(n)$, for some $n \in nodes(p)$, with the formula $\phi_n(v)$ from p_ϕ, (ii) all other nodes with T. For example, in Figure 7, $mod_S(p_{\phi 1}) = \{t_{\phi 1}\}$, $mod_S(p_{\phi 2}) = \{t'_{\phi 2}, t''_{\phi 2}\}$, $mod_S(p_{\phi 3}) = \{t_{\phi 3}\}$ and $mod_S(p_{\phi 4}) = \{t_{\phi 4}\}$.

Let t_ϕ be a decorated tree, p_ϕ a k-ary decorated pattern and S a summary. A characterization of the tuples in $p_\phi(t_\phi)$ derives directly from Proposition 3.1, considering decorated patterns and

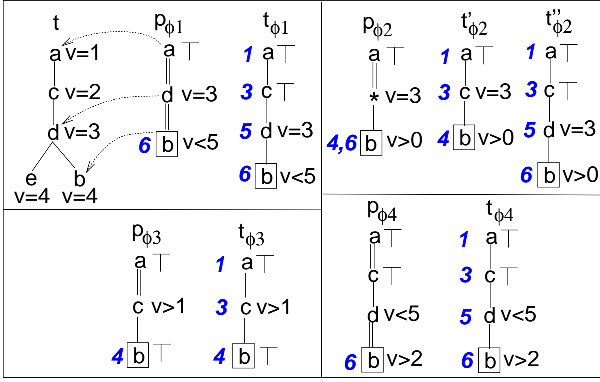


Figure 7: Decorated patterns $p_{\phi1}, p_{\phi2}, p_{\phi3}$ and $p_{\phi4}$, their canonical models, and a decorated embedding.

trees. A characterization of S-containment among decorated patterns can be similarly obtained from Proposition 4.1. Considering two decorated patterns p_ϕ, p'_ϕ and a summary S, condition 3 from Proposition 4.1 is replaced by: $\forall t_{p_\phi} \in mod_S(p_\phi)$ such that the return nodes of t_{p_ϕ} are $(n_1, \ldots, n_k)$, we have $(n_1, \ldots, n_k) \in p'_\phi(t_{p_\phi})$. For example, in Figure 7, $p_{\phi1} \subseteq_S p_{\phi2}$.

The inclusion of a decorated pattern in an union of decorated patterns can be decided along the same lines. For instance, in Figure 7, we have $p_{\phi_2} \subseteq_S p_{\phi1} \cup p_{\phi3} \cup p_{\phi4}$. The containment conditions can be found in the full version of this paper [40].

5.3 Optional pattern edges

We extend patterns to allow a distinguished subset of *optional* edges, depicted with dashed lines in patterns p_1 and p_2 in Figure 8. Pattern nodes at the lower end of a dashed edge may lack matches in a data tree, yet matches for the node at the higher end of the optional edge are retained in the pattern's semantics. For example, in Figure 8, where t is a data tree (with same-tag nodes numbered to distinguish them), $p_1(t) = \{(c_1, b_2), (c_1, b_3), (c_2, \perp)\}$, where $\perp$ denotes the null constant. Note that b_2 lacks a sibling node, yet it appears in $p_1(t)$; and, c_2 appears although it has no descendants matching d's subtree.

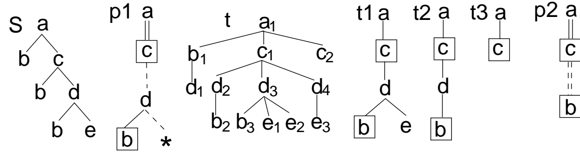


Figure 8: Optional patterns example.

Optional pattern embeddings are defined as follows. Let t be a tree and p be a pattern with optional edges. An optional embedding of p in t is a function $e : nodes(p) \to nodes(t) \cup \{\perp\}$ such that:

1. e maps the root of p into the root of t.
2. $\forall\ n \in nodes(p)$, if $e(n) \neq \perp$ and $label(n) \neq *$, then $label(n) = label(e(n))$.
3. $\forall\ n_1, n_2 \in nodes(p)$ such that n_1 is the $/$-parent (respectively, $//$-parent) of n_2:
   (a) If the edge (n_1, n_2) is not optional, then $e(n_2)$ is a child (resp. descendant) of $e(n_1)$.
   (b) If the edge (n_1, n_2) is optional: (*i*) If $e(n_1) = \perp$ then $e(n_2) = \perp$. (*ii*) If $e(n_1) \neq \perp$, let E' be the set of optional embeddings e' from the p subtree rooted at n_2, into some t subtree rooted in a child (resp. descendant) of $e(n_1)$. If $E' \neq \emptyset$, then $e(n_2) = e'(n_2)$ for some $e' \in E'$. If $E' = \emptyset$, then $e(n_2) = \perp$.

Conditions 1-3(a) above are those for standard embeddings. Condition 3(b) accounts for the optional pattern edges: we allow e to associate $\perp$ to a node n_2 under an optional edge only if no child (or descendant) of $e(n_1)$ could be successfully associated to n_2. Based on optional embeddings, optional pattern semantics is defined as in Section 3.

The canonical model $mod_S(p)$ of an optional pattern is obtained as follows. Let E be the set of optional p edges. Let p_0 be the *strict* pattern obtained from p by making all edges non-optional. For every $t_e \in mod_S(p_0)$ and set of edges $F \subseteq E$, let $t_{e,F}$ be the tree obtained from t_e by erasing all subtrees rooted in a node at the lower end of a F edge. If $p(t_{e,F}) \neq \emptyset$, add $t_{e,F}$ to $mod_S(p)$. As described, the canonical model of an optional pattern may be exponentially larger than the simple one. In practice, however, this is not the case, as Section 6 shows.

For example, in Figure 8, let p_0 be the strict pattern corresponding to p_1 (not shown in the figure), then $mod_S(p_0) = \{t_1\}$. Applying the definition above, we obtain: t_1 when $F = \emptyset$; t_2 when F contains the edge under the d node; t_3 when F contains the edge under the c node, or when F contains both optional edges. Thus, $mod_S(p_1) = \{t_1, t_2, t_3\}$.

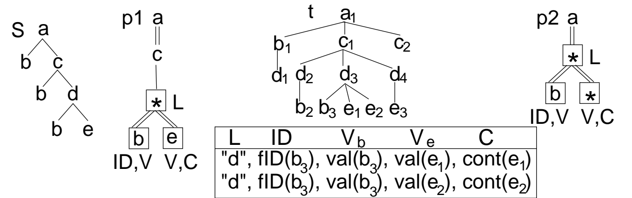
Containment for (unions of) optional patterns is determined based on canonical models as in Section 4. For example, in Figure 8, we have $p_1 \subseteq_S p_2$.

5.4 Multiple attributes per return node

So far, we have defined pattern semantics as tuples of nodes. A practical view language should allow specifying *what information items does the pattern retain from every return node*. To express this, we define *attribute patterns*, whose nodes may be annotated with up to four attributes:

- *ID* specifies that the pattern contains the node's *identifier*. The identifier is understood as an atomic value, uniquely identifying the node.
- *L* (respectively *V*) specifies that the pattern contains the node's *label* (respectively *value*).
- *C* specifies that the pattern contains the node's *content*, i.e. the subtree rooted at that node, either stored in the view, or as a reference to some repository etc. *Navigation* is possible in a C node attribute, towards the node's descendants.

In Figure 9, p_1 and p_2 are sample attribute patterns.



L	ID	V_b	V_e	C
"d",	fID(b_3),	val(b_3),	val(e_1),	cont(e_1)
"d",	fID(b_3),	val(b_3),	val(e_2),	cont(e_2)

Figure 9: Attribute pattern example.

Attribute pattern semantics is defined as follows. Let p be an attribute pattern, whose return nodes are $(n_1, \ldots, n_k)$, and t be a tree. Let $f_{ID} : nodes(t) \to \mathcal{A}$ be a labeling function assigning identifiers to t nodes. Then, $p(t, f_{ID})$ is:

$$\{\, tup(n_1, n_1^t) + \ldots + tup(n_k, n_k^t) \mid$$
$$\exists\, e : p \to t, e(n_1) = n_1^t, \ldots, e(n_k) = n_k^t \,\}$$

where $+$ stands for tuple concatenation, and $tup(n_i, n_i^t)$ is a tuple having: an attribute $ID_i = f_{ID}(n_i^t)$ if n_i is labeled ID; an attribute $L_i = label(n_i^t)$ if n_i is labeled L; an attribute $V_i = value(n_i^t)$ if n_i is labeled V; and an attribute $C_i = cont(n_i^t)$ if n_i is labeled C. For example, Figure 9 depicts $p_1(t, f_{ID})$, for the tree t and some function f_{ID}.

The S-canonical model of an attribute pattern is defined just like for regular ones. Attribute pattern containment is characterized by the same conditions as for simple patterns, *and* requires that corresponding return nodes be annotated with exactly the same attributes. In Figure 9, $p_1 \subseteq_S p_2$. Containment of unions of attribute patterns may be characterized by extending Proposition 4.2.

5.5 Nested pattern edges

We extend patterns to distinguish a subset of *nested* edges, marked by an n edge label. For example, pattern p_3 in Figure 10 is identical to p_1 in Figure 9 except for the n edge. The semantics of a nested pattern is a nested relation. Let n_1 be a pattern node and n_2 be a child of n_1 connected by a nested edge. Let n_1^t be a data node corresponding to n_1 in some data tree. The data extracted from all n_1^t descendants matching n_2 appears as a table nested inside the tuple corresponding to n_1^t. Figure 10 shows $p_3(t)$ for the tree t from Figure 9: the attributes V_e and C_e are nested under a single attribute A, corresponding to the third return node. Compare this with $p_1(t)$ in Figure 9. More details can be found in [3].

Let $p_{n,1}, p_{n,2}$ be two nested patterns whose return nodes are $(n_1^1, \ldots, n_k^1)$, respectively, $(n_1^2, \ldots, n_k^2)$, and S be a summary. For each n_i^1 and embedding $e : p_{n,1} \to S$, the *nesting sequence* of n_i^1 and e, denoted $ns(n_i^1, e)$, is the sequence of S nodes p' such that: (i) for some n' ancestor of n_i^1, $e(n') = p'$; (ii) the edge going down from n' towards n_i^1 is nested. Clearly, the length of the nesting sequence $ns(n_i^1, e)$ for any e is the number of n edges above n_i^1 in $p_{n,1}$, and we denote it $|ns(n_i^1)|$. Similar definitions hold for every n_i^2 and $e' : p_{n,2} \to S$.

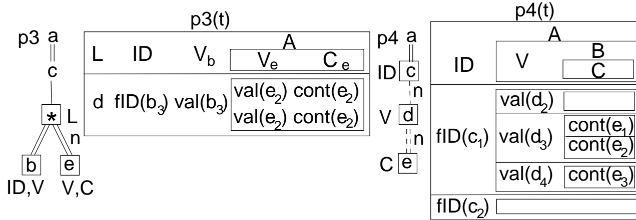


Figure 10: Nested patterns example.

PROPOSITION 5.1. Let $p_{n,1}, p_{n,2}$ be two nested patterns and S a summary as above. $p_{n,1} \subseteq_S p_{n,2}$ iff:

1. Let p_1 and p_2 be the unnested patterns obtained from $p_{n,1}$ and $p_{n,2}$. Then, $p_1 \subseteq_S p_2$.
2. For every $1 \leq i \leq k$, the following conditions hold:
   (a) $|ns(n_i^1)| = |ns(n_i^2)|$.
   (b) for every embedding $e : p_{n,1} \to S$, an embedding $e' : p_{n,2} \to S$ exists, with the same return nodes as e, such that $ns(n_i^1, e) = ns(n_i^2, e')$. ◁

Intuitively, condition 1 ensures that the tuples in $p_{n,1}$ are also in $p_{n,2}$, ignoring their nesting. Condition 2(a) requires the same nested signature for $p_{n,1}$ and $p_{n,2}$, while 2(b) requires nesting "under the same nodes" in both patterns.

Condition 2(b) can be omitted, if all edges in the nesting sequences $ns(n_i^1, e)$ and $ns(n_i^2, e')$ are one-to-one. Intuitively, nesting data under an s_1 node is the same as nesting it under its s_2 child.

Nested edges combine naturally with the other pattern extensions we presented. For example, Figure 10 shows the pattern p_2 with two nested, optional edges, and $p_4(t)$ for the tree t in Figure 8. Note the empty tables resulting from the combination of missing attributes and nested edges.

5.6 Extending rewriting

The pattern and summary extensions presented in Sections 5.1-5.5 entail, of course, that the proper canonical models and containment tests be used during rewriting. In this section, we review the remaining necessary changes to be applied to the rewriting algorithm of Section 4 to handle these extensions.

Extended summaries can be handled directly.

Decorated patterns entail the following adaptation of Algorithm 1. Whenever a join plan of the form $l_1 \bowtie_{n_1=n_2} l_2$ is considered (line 5), the plan is only built if $\phi_{n_1}(v) \wedge \phi_{n_2}(v) \neq F$, in which case, the node(s) corresponding to n_1 and n_2 in the resulting equivalent pattern(s) are decorated with $\phi_{n_1}(v) \wedge \phi_{n_2}(v)$.

Optional patterns can be handled directly.

Attribute patterns require a set of adaptations. First, some selection (σ) operators may be needed to ensure no plan is missed, as follows. Let p be a pattern corresponding to a rewriting and n be a p node. At lines 7 and 13 of the algorithm 1, we may want to test containment between q (the target pattern) and (a union involving) p. Let n_q be the q node associated to n for the containment test.

- If n is labeled $*$ and stores the attribute L (label), and n_q is labeled $l \in \mathcal{L}$, then we add to the plan associated to p the selection $\sigma_{n.L=l}$.
- If n is decorated with the formula $\phi_n(v) = T$ and stores the attribute V (value), and n_q is decorated with the formula $\phi_{n_q}(v)$, then we add to the plan associated to p the selection $\sigma_{\phi_{n_q}(v)}$.

Second, prior to Algorithm 1, we *unfold* all C attributes in the query and view patterns:

- Assume the node n in pattern p has only one associated path $s \in S$. To unfold $n.C$, we erase C and add to n a child subtree identical to the S subtree rooted in s, in which all edges are parent-child and optional, and all nodes are labeled with their label from S, and with the V attribute.
- If n has several associated paths $s_1, \ldots, s_l$, then (i) decompose p into a union of disjoint patterns such that n has a single associated path in each such pattern and (ii) unfold $n.C$ in each of the resulting patterns, as above.

Before evaluating a rewriting plan, the nodes introduced by unfolding must be extracted from the C attribute actually stored by the ancestor n. This is achieved by XPath navigation on $n.C$. Rewritings in this case will use a nav_q operator, where q is an XPath expression using the child and descendant axes, applying on the C attribute. Navigation-based rewriting is studied in [8, 27, 39].

A view pre-processing step may be enabled by the properties of the ID function f_{ID} employed in the view. For some ID functions, e.g. ORDPATHs [31] or Dewey IDs [35], $f_{ID}(n)$ can be derived by a simple computation on $f_{ID}(n')$, where n' is a child of n. If such IDs are used in a view, let $n_1 \in p_i$ be a node annotated with ID, and n_2 be its parent. Assume n_1 is annotated with the

paths $s^1_1, \ldots, s^1_k$, and n_2 with the paths $s^2_1, \ldots, s^2_l$. If the depth difference between any s^1_i and s^2_j (such that s^2_j is an ancestor of s^1_i) is a constant c (in other words, such pairs of paths are all at the same "vertical distance"), we may compute the ID of n_2 by c successive parent ID computation steps, starting from the values of $n_1.ID$.

Based on this observation, we add to n_2 a "virtual" ID attribute annotation, which the rewriting algorithm can use as if it was originally there. This process can be repeated, if n_2's parent paths are "at the same distance" from n_2's paths etc. Prior to evaluating a rewriting plan which uses virtual IDs, such IDs are computed by a special operator $nav_{f_{ID}}$ which computes node IDs from the IDs of its descendants.

Nested patterns entail the following adaptations:

First, Algorithm 1 may build, beside structural join plans (line 5), plans involving *nested structural joins*, which can be seen as simple joins followed by a grouping on the outer relation attributes. Intuitively, if a structural join combines two patterns in a large one by a new unnested edge, a nested structural join creates a new nested edge. Nested structural joins are detailed in [3, 13].

Second, prior to the containment tests, we may adapt the nesting path(s) of some nodes in the patterns produced by the rewritings. Let (l, r) be a plan-pattern pair produced by the rewriting. (i) If r has a nesting step absent from the corresponding q node, we eliminate it by applying an *unnest* operator on l. (ii) If a q node has a nesting step absent from the nesting sequence of the corresponding r node, if this r node has an ID attribute, we can produce the required nesting by a *group-by ID* operator on l; otherwise, this nesting step cannot be obtained, and containment fails.

6. EXPERIMENTAL EVALUATION

Our approach is implemented in the ULoad prototype [6]. We report on measures performed on a laptop with an Intel 2 GHz CPU and 1 GB RAM, running Linux Gentoo, and using JDK 1.5.0. We denote by XMark$n$ an XMark [38] document of n MB.

Containment To start with, we gather some statistics on summaries of several documents[2], include three snapshots of the DBLP data, from 2002, 2005 and 2006. In Table 1, n_s is the number of strong edges, and n_1 the number of one-to-one edges; such edges are quite frequent, thus many integrity constraints can be exploited by rewriting. Figure 1 demonstrates that summaries are quite small, and change little as the document grows: from XMark11 to XMark232, the summary only grows by 10%, and similarly for the DBLP data. Intuitively, the complexity of a data set levels off at some point. Thus, while summaries may have to be updated (in linear time [20]), the updates are likely to be modest.

To test containment, we first extracted the patterns of the 20 XMark [38] queries, and tested the containment of each pattern in itself under the constraints of the largest XMark summary (548 nodes). Figure 11 (top) shows the canonical model size, and containment time. Note that $|mod_S(p)|$ is small, much less than the theoretical bound of $|S|^{|p|}$. The S-model of query 7 (shown at top right in Figure 11) has 204 trees, due to the lack of structural relationships between the query variables. Such queries are not likely to be frequent in practice.

We also generated synthetic, satisfiable patterns of $3-13$ nodes, based on the 548-nodes XMark summary. Pattern node fanout is $f = 3$. Nodes were labeled $*$ with probability 0.1, and with a value predicate of the form $v = c$ with probability 0.2. We used 10 different values. Edges are labeled $//$ with probability 0.5, and are

[2]All documents, patterns and summaries used in this section are available at [36].

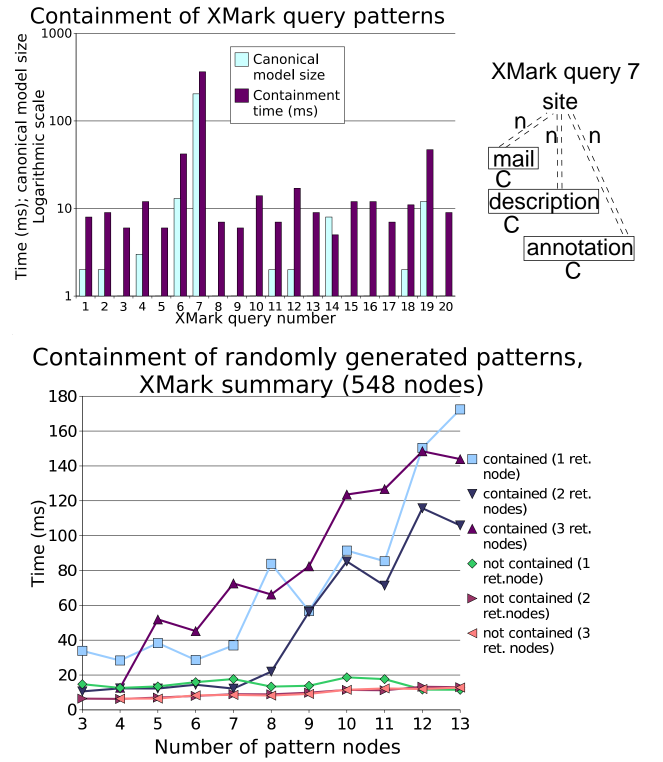


Figure 11: XMark pattern containment.

optional with probability 0.5. For this measure, we turned off edge nesting, since: randomly generated patterns with nested edges easily disagree on their nesting sequences, thus containment fails, and nesting does not significantly change the complexity (Section 5.5). For each n, we generated 3 sets of 40 patterns, having r=1, 2, resp. 3 return nodes; we fixed the labels of the return nodes to $item$, $name$, and $initial$, to avoid patterns returning unrelated nodes. For every n, every r, and every $i = 1, \ldots, 40$, we tested $p_{n,i,r} \subseteq_S p_{n,j,r}$ with $j = i, \ldots, 40$, and averaged the containment time over 780 executions. Figure 11 shows the result, separating positive from negative cases. The latter are faster because the algorithm exits as soon as one canonical model tree contradicts the containment condition, thus $mod_S(p)$ needs not be fully built. Successful test time grows with n, but remains moderate. The curves are quite irregular, since $|mod_S(p)|$ varies a lot among patterns, and is difficult to control.

We repeated the measure with patterns generated on the DBLP'05 summary. The containment times (detailed in [40]) are 4 times smaller than for XMark. This is because the XMark summary contains many nodes named *bold*, *emph* etc., thus our pattern generator includes them often in the patterns, leading to large canonical models. A query using three *bold* elements, however, is not very realistic. Such formatting tags are less frequent in DBLP's summary, making DBLP synthetic patterns closer to real-life queries. We also tested patterns with 50%, and with 0% optional edges, and found optional edges slow containment by a factor of 2 compared to the conjunctive case. The impact is much smaller than the predicted exponential worst case (Section 5.3), demonstrating the algorithm's robustness.

Rewriting We rewrite the query patterns extracted from the XMark queries [38]. The view pattern set is initialized with 2-node views, one node labeled with the XMark root tag, and the other labeled with each XMark tag, and storing ID, V, to ensure *some* rewrit-

Doc.	Shakespeare	Nasa	SwissProt	XMark11	XMark111	XMark233	DBLP '02	DBLP '06
Size	7.5 MB	24 MB	109 MB	11 MB	111 MB	233 Mb	133 MB	280 MB
$\|S\|$	58	111	264	536	548	548	145	159
$n_S(n_1)$	40 (23)	80 (64)	167 (145)	188 (153)	188 (153)	188 (153)	43 (34)	47 (39)

Table 1: Sample XML documents and their summaries.

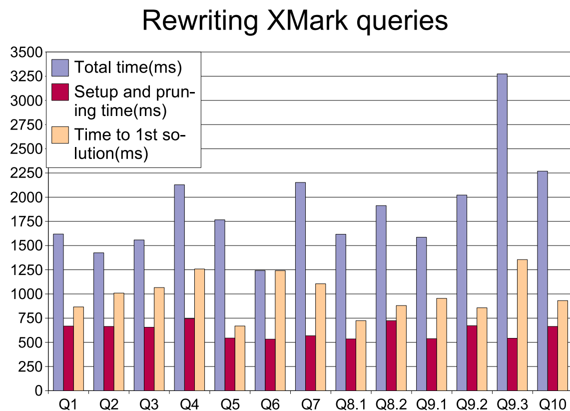


Figure 12: XMark query rewriting

ings exist. Experimenting with various synthetic views, we noticed that large synthetic view patterns did not significantly increased the number of rewritings found, because the risk that the view has little, if any, in common with the query increases with the view size. The presence of random value predicates in views had the same effect. Therefore, we then added 100 random 3-nodes view patterns based on the XMark233 summary, with 50% optional edges, such that a node stores a (*structural*) ID and V with a probability 0.75. Figure 12 shows for each query: the time to prepare the rewriting and prune the views as described in Section 4, the time elapsed until the first equivalent rewriting is found (this includes the setup time), and the total rewriting time. The first rewriting is found quite fast. This is useful since in the presence of many rewritings, the rewriting process may be stopped early. Also, view pruning was very efficient: of the 183 initial views, on average only 57% were kept.

Experiment conclusions Pattern containment performance closely tracks the canonical model size for positive tests; negative tests perform much faster. Containment performance scales up with the summary and pattern size. Rewriting performance depends on the views and number of solutions; a first rewriting is identified fast.

7. RELATED WORKS

Containment and rewriting for semistructured queries have received significant attention in the literature [19, 29, 32], in partucular under schema and other constraints [16, 17, 30, 37]. We studied pattern containment in the presence of Dataguide [20] constraints which, to our knowledge, had not been previously addressed. A summary limits tree depth (and guarantees finite algebraic rewriting), while a (recursive) schema does not. In practical documents, recursion is present, but not very deep [28], making summaries an interesting rewriting tool. More generally, schemas and summaries enable different (partially overlapping) sets of rewritings. Summary constraints are related to path constraints [11], and to the constraints used for query minimization in [2]. However, summaries describe *all* possible paths in the document, unlike the constraints of [2]. Our containment decision algorithm is related to the basic containment algorithm of [29], enhanced to benefit from summary constraints. (In the absence of a summary, our algorithm would use the canonical model of [29].) The techniques employed in [29] to speed up containment test could also be applied to our setting.

Containment of nested XQueries has been studied in [18], based on a model without node identity, unlike our model.

Query rewriting based on XPath materialized views is addressed in [8, 27, 25, 39]. Our work differs in several important respects. (i) The materialized views we consider include optional nodes, allowing them to closely fit the data needs of XQuery queries. For instance, consider the query for $x in /site//item return ⟨res⟩{$x//keyword}⟨/res⟩. The approach of [8, 27, 39] would navigate inside the view /site//item to answer. Given this view, our approach would do the same, but our view language allows specifying a much smaller view, storing the (possibly empty) nested list of keyword values for each item element, i.e. the query's data needs and nothing more. Observe that an XPath view /site//item//keyword cannot be used, since ⟨res⟩ elements must be produced even for items lacking keywords. (ii) The views we consider store *nested tuples*, allowing rewriting of queries with complex return clauses. Consider the query for $x in /site//item return ⟨res⟩⟨k⟩{$x//keyword}⟨/k⟩, ⟨p⟩{$x//price}⟨/p⟩ ⟨res⟩. No XPath view can fit tightly this query, whereas a view with two nested outerjoin edges going from an item node to the keyword and price nodes directly matches the query. (iii) Our views allow specifying interesting storage features, such as structural IDs, which increase the set of possible rewritings by allowing structural view joins. The rewritings considered in [8, 27] are limited to applying XPath navigation.

Rewriting of XQuery queries using nested XQuery views is addressed in [12, 15]. Our approach is different due to the presence of constraints (which leads to a different containment algorithm), and structural node identifiers. While using XQuery to define views seems tempting, there are some shortcomings, both noted in [15]. (i) If an XQuery view builds new elements including elements from the input, the identity of the input elements is lost (element constructors have COPY semantics). XQuery-based rewritings are thus correct as long as node identity is not an issue. We explicitly model the IDs frequently present in the store, allowing their usage in the rewriting. (ii) Consider the XQuery materialized view for $x in //item return ⟨res⟩{$x//description//keyword, $x//mail//keyword}⟨res⟩. One cannot answer //item//mail//keyword based on this view, because each item may have zero or more keywords both in its description and in the associated mailbox, and they are impossible to separate. In our approach, a view with two nested optional edges would allow answering this query.

We briefly discuss the main limitations of our approach. First, as all XPath views but unlike XQuery views, our *views* cannot invert the document nesting (e.g. store for each keyword the collection of items where it appears)[3]. This allows our views to retain their simple, easy to write tree pattern paradigm. Queries are often based on the document's original structure, thus the nesting preserved by our views is useful. We are able to rewrite *queries* inverting the document nesting. Second, our views do not store "derived" XML elements, thus, some tagging is needed for element-constructing queries. Since this is a constant-memory, linear-time operation, this is not a serious shortcoming.

Value joins are a useful feature for materialized views. Such joins can be easily supported by extending our patterns with pred-

[3]Observe that this corresponds to a group-by-value.

icates over V attributes of different nodes. Such predicates can be regarded as selections over the basic pattern; containment and rewriting are fundamentally unchanged.

In our own work, we detailed our tree pattern language in [3], and described the extraction of tree patterns from XQuery queries in [5]. The algorithm computing the equivalent pattern to a given plan (part of the rewriting algorithm in Section 4) is detailed in [4]. The ULoad system architecture was outlined in a four-pages demo proposal [6]. We study some optimization techniques enabled by structural summaries in XML query processing (excluding containment or rewriting) in [7]. A preliminary version of this paper has been presented in *Journées de Bases de Données Avancées* 2006, an informal event without proceedings or ISBN.

8. CONCLUSION

We studied the problem of XML query pattern rewriting based on summary constraints, using detailed information about view contents and interesting properties of element IDs. Each of these features enables rewritings which would not otherwise be possible. Our future work includes extending ULoad with XML Schema constraints, and view maintenance in the presence of updates.

9. REFERENCES

[1] 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.

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

[3] A. Arion, V. Benzaken, and I. Manolescu. XML Access Modules: Towards Physical Data Independence in XML Databases. XIMEP Workshop, 2005.

[4] A. Arion, V. Benzaken, I. Manolescu, and Y. Papakonstantinou. Trading with plans and patterns in XQuery optimization. Tech. report, available at www-rocq.inria.fr/~arion, 2006.

[5] A. Arion, V. Benzaken, I. Manolescu, Y. Papakonstantinou, and R. Vijay. Algebra-based identification of tree patterns in XQuery. In *Flexible Query Answering Systems*, 2006.

[6] A. Arion, V. Benzaken, I. Manolescu, and R. Vijay. ULoad: Choosing the Right Store for your XML Application (demo). In *VLDB*, 2005.

[7] A. Arion, A. Bonifati, I. Manolescu, and A. Pugliese. Path summaries and path partitioning in modern XML databases. *Currently under revision for WWW Journal*.

[8] A. Balmin, F. Ozcan, K. Beyer, R. Cochrane, and H. Pirahesh. A framework for using materialized XPath views in XML query processing. In *VLDB*, 2004.

[9] P. Bohannon, J. Freire, P. Roy, and J. Simeon. From XML schema to relations: A cost-based approach to XML storage. In *ICDE*, 2002.

[10] P. Boncz, T. Grust, M.van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *SIGMOD*, 2006.

[11] P. Buneman, W. Fan, and S. Weinstein. Interaction between path and type constraints. *ACM Trans. Comput. Log*, 4(4), 2003.

[12] L. Chen and E. Rundensteiner. XCache: Xquery-based caching system. In *WebDB*, 2002.

[13] Z. Chen, H.V. Jagadish, L. Lakshmanan, and S. Paparizos. From tree patterns to generalized tree patterns: On efficient evaluation of XQuery. In *VLDB*, 2003.

[14] B. Cooper, N. Sample, M. Franklin, G. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *VLDB*, 2001.

[15] A. Deutsch, E. Curtmola, N. Onose, and Y. Papakonstantinou. Rewriting nested XML queries using nested XML views. In *SIGMOD*, 2006.

[16] A. Deutsch and V. Tannen. Containment and integrity constraints for XPath. In *KRDB Workshop*, 2001.

[17] A. Deutsch and V. Tannen. MARS: A system for publishing XML from mixed and redundant storage. In *VLDB*, 2003.

[18] X. Dong, A. Halevy, and I. Tatarinov. Containment of nested XML queries. In *VLDB*, 2004.

[19] D. Florescu, A. Levy, and D. Suciu. Query containment for conjunctive queries with reg. expressions. In *PODS*, 1998.

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

[21] G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *PODS*, 2003.

[22] T. Grust, M. van Keulen, and J. Teubner. Accelerating XPath evaluation in any RDBMS. *TODS*, 29, 2004.

[23] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. Lakshmanan, A. Nierman, S. Paparizos, J. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. Timber: A native XML database. *VLDB J.*, 11(4), 2002.

[24] H. Jiang, H. Lu, W. Wang, and J. Xu. XParent: An efficient RDBMS-based XML database system. In *ICDE*, 2002.

[25] L. Lakshmanan, H. Wang, and Z. Zhao. Answering tree pattern queries using views. In *VLDB*, 2006.

[26] A. Levy, A. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *PODS*, 1995.

[27] B. Mandhani and D. Suciu. Query caching and view selection for XML databases. In *VLDB*, 2005.

[28] L. Mignet, D. Barbosa, and P. Veltri. The XML web: A first study. In *Proc. of the Int. WWW Conf.*, 2003.

[29] G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *PODS*, 2002.

[30] F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *ICDT*, 2003.

[31] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-friendly XML node labels. In *SIGMOD*, 2004.

[32] Y. Papakonstantinou and V. Vassalos. Query rewriting for semistructured data. In *SIGMOD*, 1999.

[33] C. Qun, A. Lim, and K. Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *SIGMOD*, 2003.

[34] J. Shanmugasundaram, H. Gang, K. Tufte, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB*, 1999.

[35] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD*, 2002.

[36] ULoad Web site. gemo.futurs.inria.fr/projects/XAM.

[37] P. Wood. Containment for XPath fragments under DTD constraints. In *ICDT*, 2003.

[38] The XMark benchmark. www.xml-benchmark.org, 2002.

[39] W. Xu and M. Ozsoyoglu. Rewriting XPath queries using materialized views. In *VLDB*, 2005.

[40] Full version of this paper. INRIA HAL report 1233, available at hal.inria.fr, 2006.