Structured Materialized Views for XML Queries

Ioana Manolescu[1] Véronique Benzaken[2] Andrei Arion[1]
Yannis Papakonstantinou[3]

[1] INRIA Futurs - 4 rue J. Monod, 91893 Orsay, France - firstname.lastname@inria.fr
[2] LRI - Bat 490 Universite Paris-Sud 91405 Orsay, France - firstname.lastname@lri.fr
[3] UCSD - 9500 Gilman Drive, La Jolla, CA 92093, USA - firstname@cs.ucsd.edu

Abstract

The performance of XML database queries can be greatly enhanced by employing materialized views. We present containment and rewriting algorithms for tree pattern queries that correspond to a large and important subset of XQuery, in the presence of a structural summary of the database (i.e., in the presence of a Dataguide). The tree pattern language captures structural identifiers and optional nodes, which allow us to translate nested XQueries into tree patterns. We characterize the complexity of tree pattern containment and rewriting, under the constraints expressed in the structural summary, whose enhanced form also entails integrity constraints. Our approach is implemented in the ULoad [5] prototype and we present a performance analysis.

Keywords: XML, XQuery materialized views, query processing

1 Introduction

Materialized views can greatly improve query processing performance. 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. We study the problem of rewriting a query using materialized views, whereas both the query and the views are described by a tree pattern language, which is appropriately extended to capture a large XQuery subset. We assume the presence of a structural summary and structural identifiers; both increase the opportunities for rewriting.

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

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

A simplified XMark document fragment appears in Figure 1(a). 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 rewriting opportunities. In short, a structural summary (or strong Dataguide [17]) of an XML document is a tree, including all paths occurring in the document. Figure 1(b) shows the structural summary of the document in Figure 1(a).

Each view is defined by an extended tree pattern and produces a nested table, which may include null values. Figure 1(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 ($\perp$).

Furthermore, if an edge is labeled as n, 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.

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 1(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.

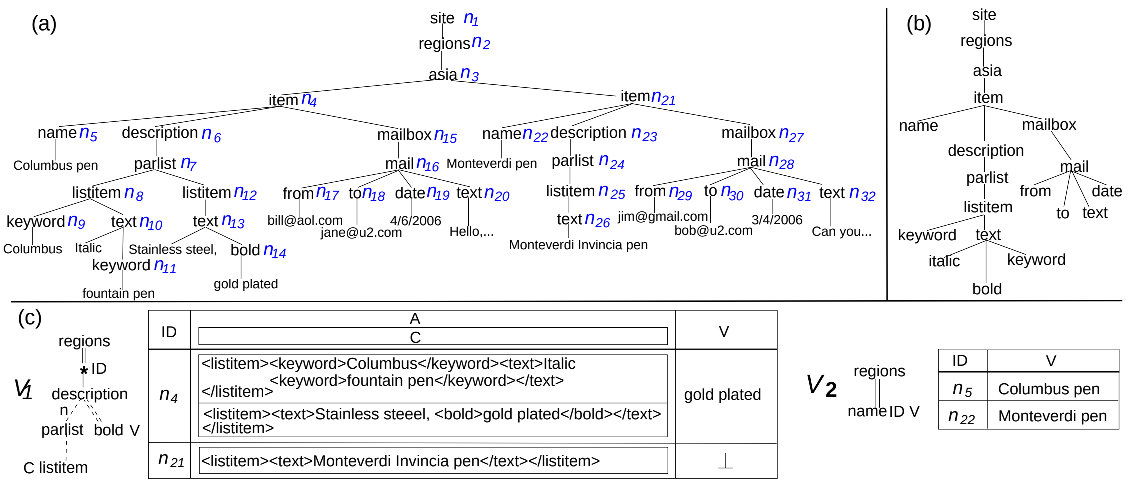

Figure 1: (a) XMark document fragment, (b) its structural summary and (c) two materialized views.

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 rewriting 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 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 require 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 [17]. The XMark DTD [31] 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 [20]. 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, 23, 27]. 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 [23, 27]. Assuming V_1 stored the ID of ⟨parlist⟩ nodes, we could derive from it the ID of their parent ⟨description⟩ nodes, and 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.

Contributions and outline We address the problem of view-based XML query containment and rewriting 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 [6, 32] and tree pattern [2, 10, 25] formalisms. Given a query and a set of views:

- We characterize the complexity of pattern containment under Dataguides [17] and integrity constraints, and provide a containment decision algorithm.
- We describe a sound and complete view-based rewriting algorithm which produces an algebraic plan combining the tree pattern views, whose result is, for all inputs, equivalent to the query result in the presence of Dataguide constraints.
- The containment and rewriting algorithms have been fully implemented in the ULoad prototype, which was recently demonstrated [5]. We report on their practical applicability and performance.

The novelty of our work is manifold. (*i*) Going beyond XPath views [6, 32], our tree patterns store data for several nodes, feature optional and/or nested edges, and describe interesting ID properties, crucial for the success of rewriting. (*ii*) To the best of our knowledge, ours is the first work to address XML query rewriting under Dataguide constraints. Strong Dataguides can be built and maintained in linear time out of tree-structured data [17]. Our experimental observations confirm those of [17], demonstrating that in many practical applications, Dataguides are very compact, and can be efficiently exploited.

This paper is organized as follows. Section 2 reviews preliminary definitions. For readability, containment and rewriting algorithms are presented in two steps. Section 3 considers containment and rewriting for a very simple flavor of conjunctive patterns and constraints, while Section 4 extends these results to the full tree

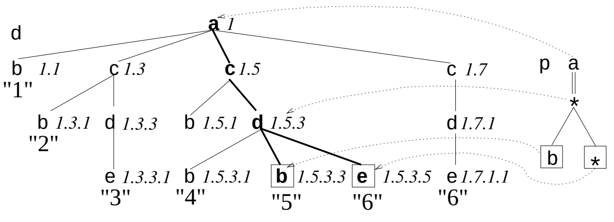

Figure 2: Sample XML document d, conjunctive pattern p, and embedding $e : p \to d$.

pattern language and to richer constraints. Section 5 presents a performance evaluation. We review related works, and conclude.

2 Preliminaries

2.1 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 XML 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 2 (left) depicts a sample XML document, where node values are shown underneath the node label, e.g. "1", "2" etc. Other notations in Figure 2 will be explained shortly.

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 $n_1 \prec\!\!\prec n_2$.

2.2 Conjunctive tree patterns

We recall the classical notions of conjunctive tree patterns and embeddings [2, 21]. 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. At right in Figure 2, we show a pattern p, whose return nodes are enclosed in boxes.

An *embedding* of a conjunctive tree pattern p into an XML 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 $n_1 \prec n_2$, $e(n_1) \prec e(n_2)$.
- For any $n_1, n_2 \in nodes(p)$ such that $n_1 \prec\!\!\prec n_2$, $e(n_1) \prec\!\!\prec e(n_2)$

Dotted arrows in Figure 2 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 subtree of t 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 2, the t subtree shown in bold is $e(p)$. We may use the notation $u \in e(p)$ to denote that node u appears in the tree $e(p)$. In Figure 2, note that $e(p)$ contains has more nodes than p, since the intermediary c node also belongs to $e(p)$.

2.3 Path summaries

Given a document d, a *rooted simple path* (or simply *path*) is a succession of /-separated labels

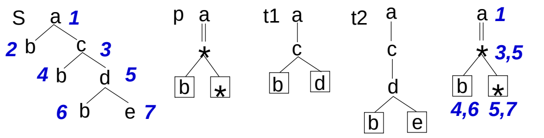

Figure 3: Summary S of the document in Figure 2, pattern p, $mod_S(p) = \{t_1, t_2\}$, and annotated p.

$/l_1/l_2/\ldots/l_k$, $k \leq 1$, such that l_1 is the label of d's root, l_2 is the label of one of the root's children, l_3 the label of one node on the path $/l_1/l_2$ etc. Note that only node labels (not values) appear in paths.

The *simple 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 nodes $n_1, n_2, \ldots, n_k \in d$ reachable by the same path p from d's root to the same node $n_p \in S(d)$. We may use a path p to designate its corresponding node in $S(d)$. The parent $\prec$ and descendent $\prec\!\!\prec$ notations extend naturally to summary nodes. Figure 3 (left) shows the summary corresponding to the document in Figure 2.

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

2.4 Summary-based canonical model

Let p be a conjunctive tree pattern, and S be a summary. The *S-canonical model of p*, denoted $mod_S(p)$, is the set of all S subtrees $t_e = e(p)$ where e is an embedding $e : p \to S$. 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.

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

Proposition 2.1 *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 (although strictly speaking, t_e is a subtree of S, and thus disjoint from 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.*

The proof of the proposition can be found in the extended version of this paper ([19]).

For example, in Figure 2, 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 2.1 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 2.1 *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 $e : p \to S$, $e(n) = s_n$.*

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.

3 Summary-based containment and rewriting of conjunctive patterns

3.1 Summary-based containment

We start by defining pattern containment under summary constraints:

Definition 3.1 *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 $S \models t$, $p(t) \subseteq p'(t)$.*

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

Proposition 3.1 *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 $(n_1^t, \ldots, n_k^t) \in p'(t_p)$.*

Proposition 3.1 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 $O(|mod_S(p)| \times |S| \times |p'|)$, since each $mod_S(p)$ tree has at most $|S|$ nodes, and $p'(t_e)$ can be computed in $|t_e| \times |p'|$. In the worst case, $|mod_S(p)|$ is $|S|^{|p|}$, however in practice the size is much smaller, as we show in Section 5.

A simple extension of Proposition 3.1 addresses containment for unions of patterns:

Proposition 3.2 *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 $(n_1, \ldots, n_k) \in p'_i(t_e)$.*

The proof can be found in the extended version of this paper ([19]). We define *S-equivalence* as two-way containment, and denote it $\equiv_S$. When S is obvious from the context, we simply call it equivalence.

3.2 Summary-based rewriting

Let $p_1, \ldots, p_n$ and q be some patterns and S be a summary. The problem of *rewriting q using $p_1, \ldots, p_n$ under S constraints* consists of finding all algebraic expressions e built with the patterns p_i and the operators $\cup, \bowtie_=, \bowtie_\prec, \bowtie_{\prec\prec}$, and π, such that $e \equiv_S q$. Here, $op_1 \bowtie_= op_2$ denotes a join pairing input tuples which contain exactly the same node, while $\bowtie_\prec, \bowtie_{\prec\prec}$ denote structural joins returning tuples where nodes from one input are parent/ancestors of nodes from the other input. Note that we are interested in *logical algebraic expressions*, which we will simply call *plans.*

Clearly, the plans of two rewritings may syntactically differ, while being equivalent by virtue of well-known algebraic laws (thus, clearly, also S-equivalent), such as $\pi_{n_1}(\pi_{n_1,n_2}(p))$ and $\pi_{n_1}(p)$. One could obtain such a plan from the other by applying those laws. Therefore, we reformulate the problem into: *find all plans e (up to algebraic equivalence) such that $e \equiv_S q$.*

A simple rewriting algorithm consists of building plans based on $p_1, \ldots, p_n$, and testing their S-equivalence to the target pattern q. However,

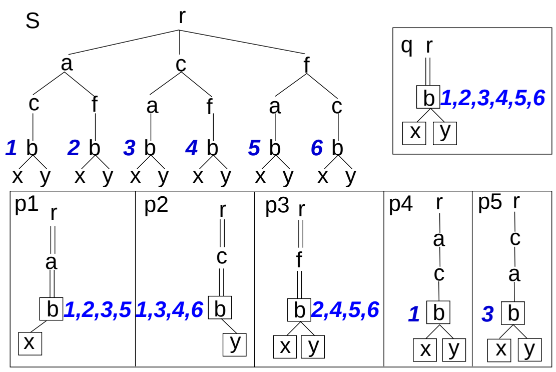

Figure 4: Summary S, query q and patterns p_1-p_5.

it is not clear how to test equivalence between plans and patterns under summary constraints. In contrast, we do have a containment decision algorithm for conjunctive patterns.

This leads to the idea of manipulating, during rewriting, *plan-pattern pairs*, such that in each pair, the plan and the pattern are by construction S-equivalent. A plan is equivalent to the query q *iff* the pattern associated to the plan is equivalent to q.

Note, however, that not any plan has an equivalent pattern, as illustrated in Figure 4. The S paths associated to the b pattern nodes are shown next to the nodes. The only S-equivalent rewriting of q based on p_1, p_2, p_3 is $(p_1 \bowtie_{b=b} p_2) \cup p_3$, yet no pattern is 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 descendent of c in the hypothetic pattern equivalent to $p_1 \bowtie_{b=b} p_2$. However, $(p_1 \bowtie_{b=b} p_2) \equiv_S (p_4 \cup p_5)$, where p_4, p_5 are the patterns at right in Figure 4. More generally:

Proposition 3.3 *Any algebraic plan built with* $\bowtie_=, \bowtie_\prec, \bowtie_{\prec\!\prec}$*, and* π *on top of some patterns* $p_1, \ldots, p_n$ *is* S*-equivalent to a union of conjunctive patterns.*

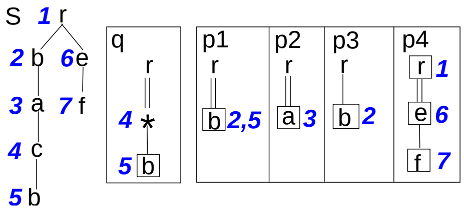

Figure 5: Sample configuration for pattern joins.

In practive, the situations where unions are actually required to get an equivalent representation of a join result are not very frequent.

Traditionally, the rewriting of a conjunctive relational query is driven by the query itself. For instance, the bucket algorithm [18] collects possible rewritings for every query atom, and builds complete rewritings by combining them. A rewriting exists *iff* there are rewritings for every atom, and if they can be combined. An interesting question is, then, whether such target query-driven techniques may be used in our case. In other words, can we rewrite q by finding rewritings for every q node and then combining them ?

The answer is no: finding rewritings for every q node is neither sufficient, nor necessary. To see that it is not necessary, consider, for instance, 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).

To see that covering all q nodes is not sufficient, consider Figure 5, where 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 $(p_2 \bowtie_{a \prec\!\prec b} p_1) \equiv_S q$, thus the rewriting process must explore such plans.

Algorithm 1: Conjunctive pattern rewriting under summary constraints

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

$M_0 \leftarrow \{(p_i, p_i) \mid 1 \leq i \leq n\}$; $M \leftarrow M_0$
repeat
  foreach $(l_i, p_i) \in M, (l_j, p_j) \in M_0$ **do**
    foreach *possible way of joining* l_i *and* l_j *using* $\bowtie_{=_{id}}, \bowtie_{\prec}, \bowtie_{\prec\!\prec}$ **do**
      $(l, p) \leftarrow (l_i, p_i) \bowtie (l_j, p_j)$
      if $p \neq p_i$ *and* $p \neq p_j$ **then**
        if $p \equiv_S q$ **then**
          output l
        else
          if $|l| \leq |q| \times |S|$ **then**
            $M \leftarrow M \cup \{(l, p)\}$
until M *is stationary*
foreach *minimal* $N \subseteq M$ *s.t.* $\cup_{(l,p) \in N}\, p \equiv_S q$ **do**
  output $\cup_{(l,p) \in N}\, p$

In contrast with relational query rewriting, an equivalent rewriting of a conjunctive pattern under S constraints may be a union of plans. For example, considering p_1 in Figure 5 as the query, a possible rewriting is $q \cup p_3$.

In practice, one is typically interested in *minimal* rewritings only, that is, plans such that no subplan thereof is a rewriting. Let S be a summary, and assume we are rewriting q using $p_1, \ldots, p_n$. The following two propositions allow restricting the search to avoid non-minimal rewritings:

Proposition 3.4 *Assume that for some* $1 \leq i \leq n$*, 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 descendent of* y*. Let* e *be a rewriting of* q *in which* p_i *appears. Then there exists a rewriting* e' *which is a subplan of* e*, but* e' *does not use* p_i*.*

The data contained in such a pattern p_i belongs to different parts of the document than those needed by the query, thus p_i can be discarded. An example is pattern p_4 for the rewriting of q in Figure 5.

Proposition 3.5 *Assume that for some plan-pattern pairs* (l_i, r_i) *and* (l_j, r_j) *and possible join result* $(l, r) = (l_i, r_i) \bowtie (l_j, r_j)$*, the patterns (or pattern sets)* r *and* r_i *coincide (in their tree structure and associated paths). Let* e *be a* q *rewriting using* (l, r)*. Then there exists a rewriting* e' *which is a subplan of* e *but which uses* l_i *instead of* l*.*

Proposition 3.5 allows to avoid building a (plan, pattern) pair, if the resulting pattern does not differ from the pattern of one of its children. Intuitively, such a (plan, pattern) pair does not open any new rewriting possibilities.

The following proposition limits the size of the join plans explored:

Proposition 3.6 *Given a pattern* q *and summary* S*, the size of a join plan* p*, part of a minimal rewriting of* q*, is at most* $|q| \times |S|$*, where* $|q|$ *is the number of* q *nodes and the size of* p *is the number patterns* p_i *appearing in* p*.*

The intuition is that an equivalent rewriting has to enforce the structural relationships between all q nodes. Enforcing each q edge may require joining at most $|S|$ patterns.

Prior to testing whether a pattern p obtained via rewriting is S-contained in q, one must identify k return nodes of p, namely $n_1, \ldots, n_k$, where k is the arity of q, extract from p a pattern p' whose only return nodes are $n_1, \ldots, n_k$, then test if $p' \subseteq_S q$. This choice of k nodes is needed because containment is defined on same-arity patterns. If p's arity is smaller than k, clearly $p \not\equiv_S q$. Otherwise, there are many ways of chosing k return nodes of p, which may lead to a large number of containment tests.

The following proposition allows to significantly reduce these tests:

Proposition 3.7 *Let p, q be two k-ary patterns and S a summary. 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 are a subset of the S paths associated to m_i.*

3.3 Rewriting algorithm

Algorithm 1 describes conjunctive pattern rewriting. M_0 is the set of initial (p_i, p_i) pairs, where the first p_i is interpreted as a plan, and the second as a pattern. We assume the set $p_1, \ldots, p_n$ is pruned according to Proposition 3.4 prior to running Algorithm 1. M is the working set, initialized at M_0; intermediary plans accumulate in M. Join plans are developed at lines 2-11; we build left-deep plans only (the right-hand join operand comes from M_0), to avoid constructing rewritings which differ only by their join orders. As soon as (l, p) is obtained, p's satisfiability is tested, and if p is S-unsatisfiable, (l, p) is discarded. The condition at line 6 derives from Proposition 3.5.

Union plans are built on top of join plans at lines 13-14 (obviously, the two could have been intertwined). The set N is minimal in the sense that for any $N' \subset N$, $\cup_{(l,p) \in N'} p$ is not an equivalent rewriting of q.

The $\equiv_S$ tests (lines 7 and 13) are performed based on Propositions 3.1 and 3.2. When looking for ways of choosing k return nodes prior to the containment test (lines 7, 13), thanks to Proposition 3.7 we only consider those $(n_1, \ldots, n_k)$ tuples of p return nodes such that the paths associated to each return node n_i are a subset of the paths associated to the corresponding q return node.

The condition at line 10 guards the addition of a new (plan, pair) to the working set, according to Proposition 3.5.

Proposition 3.8 *Algorithm 1 is correct and complete. It produces all $\equiv_S$ minimal rewritings of q (up to algebraic equivalence) based on $p_1, \ldots, p_n$, under S constraints.*

The complexity of Algorithm 1 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 using a join plan).

4 Complex summaries and patterns

In this section, we 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 4.1), respectively, more complex patterns. Section 4.2 considers patterns endowed with value predicates, Section 4.3 addresses patterns with optional edges, Section 4.4 describes containment of patterns which may store several data items for a given node, and Section 4.5 enriches patterns with nested edges. Finally, Section 4.6

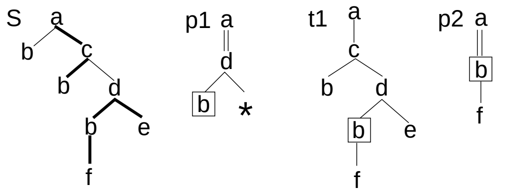

Figure 6: Enhanced summary and sample patterns.

outlines the impact of these extensions on the rewriting algorithm.

4.1 Enhanced summaries

Useful information for the rewriting process may be derived from an *enhanced summary*, or summaries with integrity constraints. Let d be a document and S_0 be its (plain) summary. Its enhanced summary S is obtained from S_0 by distinguishing a set of edges as *strong*. 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 said *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 either 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.

The notion of *conforming to a summary* naturally extends to enhanced summaries. A document d conforms to an enhanced summary S iff d conforms to the simple summary S_0 obtained from S, and furthermore, d respects the parent-child integrity constraints enforced by strong S edges. Pattern containment based on enhanced summary constraints can then be defined.

The difference between simple and enhanced summaries is visible at the level of canonical models. Let S be an enhanced summary, and p a conjunctive pattern. The canonical model of p based on S, 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 connecting them to their parents in S.

Modulo the modified canonical model, enhanced summary-based containment can be decided just like for simple summaries. For example, applying Proposition 3.1 in Figure 6, we obtain that patterns p_1 and p_2 are S-equivalent.

4.2 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 $\wedge$.

Containment on (unions of) decorated patterns requires some (space-consuming, yet conceptually simple) extensions that are detailed in [19].

4.3 Optional pattern edges

We extend patterns to allow a distinguished subset of *optional* edges, depicted with dashed lines; p_1 and p_2 in Figure 7 illustrate this. Intuitively, 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 7, 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.

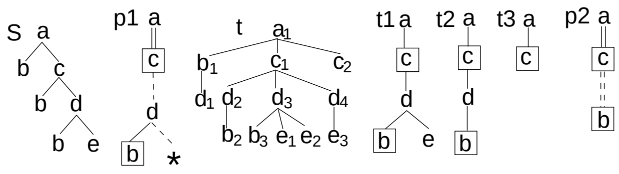

Figure 7: Optional patterns example.

To formally define semantics of optional patterns, we introduce optional embeddings.

Definition 4.1 *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. descendent) 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. descendent) 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 descendent) of $e(n_1)$ could be successfully associated to n_2.

Based on optional embeddings, optional pattern semantics is defined as in Section 2.2.

Given a summary S and an optional pattern p, $mod_S(p)$ 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 $mod_S(p)$.

For example, in Figure 7, 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; 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, $mod_S(p_1) = \{t_1, t_2, t_3\}$.

As described above, 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 5 shows.

Containment for (unions of) optional patterns is determined based on canonical models as in Section 3. For example, in Figure 7, we have $p_1 \subseteq_S p_2$.

4.4 Multiple attributes per return node

So far, we have defined pattern semantics abstractly as tuples of nodes. For practical reasons, however, one should be able to specify *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. The subtree may be stored in a compact encoding, or as a reference to some repository etc. We will only retain that a *navigation* is possible in a *C* node attribute, towards the node's descendents.

Figure 8 depicts the attribute patterns p_1 and p_2.

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

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

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 8 depicts $p_1(t, f_{ID})$, for the data tree t and some labeling function f_{ID}.

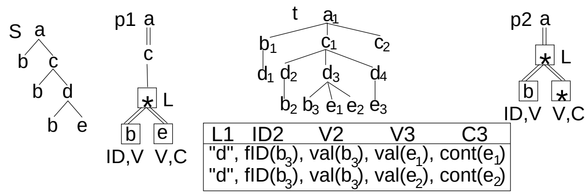

Figure 8: Attribute patterns.

The S-canonical model of an attribute pattern is defined just like for regular ones. Attribute pattern containment is characterized as follows:

Proposition 4.1 *Let $p_{1,a}, p_{2,a}$ be two attribute 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. We have $p_{1,a} \subseteq_S p_{2,a}$ iff:*

1. *For every i, $1 \leq i \leq k$, node n_i^1 is labeled ID (respectively, V, L, C) iff node n_i^2 is labeled ID (respectively, V, L, C).*

2. *Let p_2 be the simple pattern obtained from $p_{2,a}$. For every $t_e \in mod_S(p_{1,a})$, whose return nodes are $(n_1^t, \ldots, n_k^t)$, we have $(n_1^t, \ldots, n_k^t) \in p_2(t_e)$.*

In Figure 8, $p_1 \subseteq_S p_2$. Containment of unions of attribute patterns may be characterized by extending Proposition 3.2 with a condition similar to 1 above.

4.5 Nested pattern edges

We extend our patterns to distinguish a subset of *nested* edges, marked by an n edge label. See, for example, pattern p_3 in Figure 9, identical to

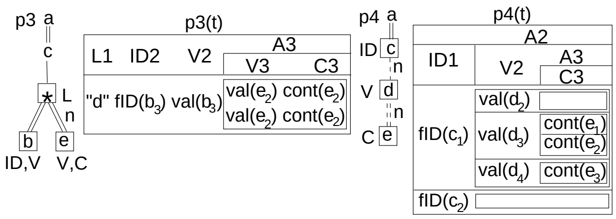

Figure 9: Nested patterns and their semantics.

p_1 in Figure 8 except for the n edge[1]. 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 descendents matching n_2 will appear as a grouped table inside the single tuple corresponding to n_1^t. Figure 9 shows $p_3(t)$ for the tree t from Figure 8. Here, the attributes V_3 and C_3 have been nested under a single attribute A_3, corresponding to the third return node. Compare this with $p_1(t)$ in Figure 8. The semantics of a nested pattern is a nested relation (detailed 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 $|ns(n_i^1)|$. For every n_i^2 and $e' : p_{n,2} \to S$, the nesting sequence $ns(n_i^2, e')$ is similarly defined.

[1]Edge nesting and node attributes are, of course, orthogonal features. We used a nested *attribute* pattern in Figure 9 solely to ease comparison with Figure 8.

Proposition 4.2 *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 $e : p_{n,1} \to S$, there exists an embedding $e' : p_{n,2} \to S$ 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_1 are also in p_2, abstraction being made from their nesting. Condition 2(a) requires the same nested signature for p_1 and p_2, while 2(b) imposes that nesting be applied "under the same nodes" in both patterns.

Condition 2(b) can be safely relaxed, in the presence of 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. Then, nesting data under an s_1 node has the same effect as nesting it under its s_2 child. Taking into account such information, the equality in condition 2(b) is replaced by: $ns(n_i^1, e)$ and $ns(n_i^2, e')$ are connected by one-to-one edges only.

Nested edges combine naturally with the other pattern extensions we presented. For example, Figure 9 shows the pattern p_2 with two nested, optional edges, and $p_4(t)$ for the tree t in Figure 7. Note the empty tables resulting from the combination of missing attributes and nested edges.

4.6 Extending rewriting

The pattern and summary extensions presented in Sections 4.1-4.5 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 3.3 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, we need to refine Proposition 3.5 to consider two patterns equal if their nodes and associated paths are the same *and* if their attribute annotations are the same. For instance, when rewriting the query $q = //* \, ID \, L \, V$, if $p_1 = //* \, ID \, L$ and $p_2 = //* \, ID \, V$, the join $p_1 \bowtie_{ID=ID} p_2$ is useful, because the resulting pattern has more attributes than p_1 or p_2, even if its nodes and paths are the same as those of p_1 and p_2.

Second, 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 $\sigma_{\phi_{n_q}(v)}$.

Third, 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 stored in the (unfolded) ancestor n. This is achieved by XPath navigation on $n.C$.

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 [23] (illustrated in Figure 2) or Dewey IDs [27], $f_{ID}(n)$ 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 $s_1^1, \ldots, s_k^1$, and n_2 with the paths $s_1^2, \ldots, s_l^2$. If the depth difference between any s_i^1 and s_j^2 (such that s_j^2 is an ancestor of s_i^1) 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 descendents.

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 entails a new nested one. Nested structural joins are detailed in [3, 10].

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* operator on l; otherwise, this nesting step cannot be obtained, and containment fails.

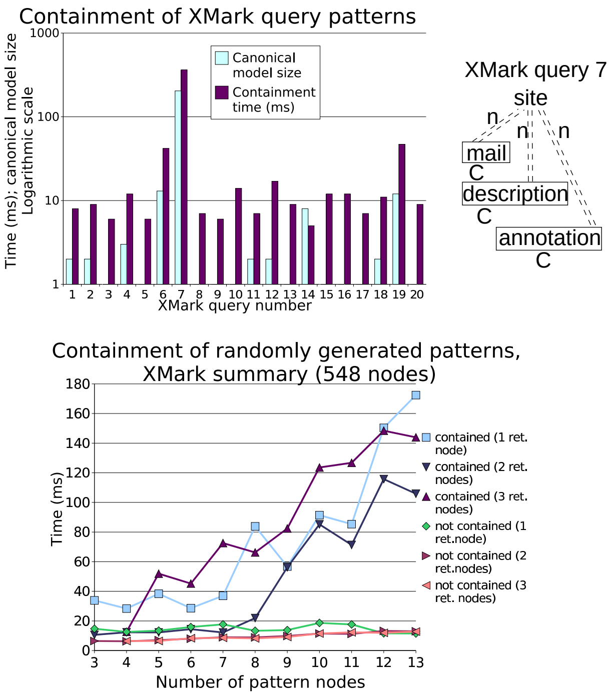

Figure 10: XMark pattern containment.

5 Experimental evaluation

Our approach is implemented in the ULoad Java-based prototype [5, 28]. We report on measures performed on a laptop with an Intel 2 MHz CPU and 1 GB RAM, running Linux Gentoo, and using JDK 1.5.0. We denote by XMark$n$ an XMark [31] document of n MB. ***All documents, patterns and summaries used in this section are available at [28].***

Containment To start with, we gather some statistics on summaries of several documents, 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

Doc.	Shakespeare	Nasa	SwissProt	XMark11	XMark111	XMark233	DBLP '02	DBLP '05	DBLP '06
Size	7.5 MB	24 MB	109 MB	11 MB	111 MB	233 Mb	133 MB	280 MB	326 MB
$\|S\|$	58	24	117	536	548	548	145	159	164
$n_S(n_1)$	40 (23)	80 (64)	167 (145)	188 (153)	188 (153)	188 (153)	43 (34)	47 (39)	

Table 1: Sample XML documents and their summaries.

many integrity constraints can be exploited by rewriting. Table 1 demonstrates 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 [17]), the updates are likely to be modest.

To test containment, we first extracted the patterns of the 20 XMark [31] queries, and tested the containment of each pattern in itself under the constraints of the largest XMark summary (548 nodes). Figure 10 (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 10) has 204 trees, due to the lack of structural relationships between the query variables, which is not the frequent case in practice. The impact of optional edges on the canonical model size is quite moderate: 16 XMark patterns have optional edges, yet small canonical models (except for query 7).

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 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 4.5). 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 10 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 $mod_S(p)$ need 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'06 summary. The containment times (detailed in Figure 11) 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 sum-

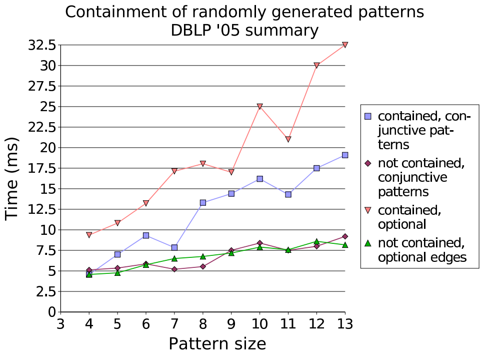

Figure 11: DBLP pattern containment.

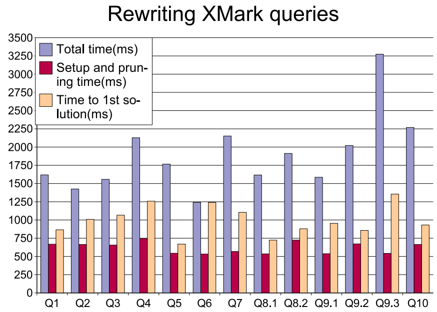

Figure 12: XMark query rewriting

mary, 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 4.3), demonstrating the algorithm's robustness.

Rewriting We rewrite the query patterns extracted from the XMark [31]. 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* rewritings 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 generated 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 3, 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.

6 Related works

Containment and rewriting for semistructured queries have received significant attention in the literature, either in the general case [16, 24, 21], or under schema and other semantic constraints [13, 14, 22, 30]. We studied tree pattern containment in the presence of Dataguide [17] constraints which, to our knowledge, had not been previously addressed. One difference between schema and summary constraints is that 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 [20], making summaries an interesting rewriting tool. More generally, schemas and summaries enable different (partially overlapping) sets of rewritings. Our containment decision algorithm is related to the basic containment algorithm of [21], enhanced to benefit from summary constraints. The optimizations proposed in [21] could also be applied to our setting, speeding up containment. Summary constraints are related to path constraints [8], and to the constraints used for query minimization in [2]. However, summaries allow describing *all* possible paths in the document, which the constraints of [2] do not.

An algebraic framework for unconstrained XQuery minimization is described in [12]. Containment of nested XQueries has been studied in [15], based on a model without node identity, unlike our model.

Recent works have addressed materialized view-based XML query rewriting [6, 9, 11, 32]. The novelty of our work consists on using summary constraints, and information about the view attributes and their interesting properties useful for rewriting. Restricted to unnested views, our rewriting problem bears similarities with the problem of answering XQuery queries when the data is shredded in a relational database, studied e.g. in [26]. However, our approach does not need SQL as an intermediary language.

The patterns we consider are similar to those of [10, 25], which, however, did not consider view-based rewriting.

7 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; all these features tend to enable 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.

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, Y. Papakonstantinou, and R. Vijay. Algebra-based identification of tree patterns in XQuery. In *FQAS*, 2006.

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

[6] 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.

[7] A. Barta, M. Consens, and A. Mendelzon. Benefits of path summaries in an XML query optimizer supporting multiple access methods. In *VLDB*, 2005.

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

[9] L. Chen, E. Rundensteiner, and S. Wang. XCache: a semantic caching system for XML queries (demo). In *SIGMOD*, 2002.

[10] 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.

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

[12] A. Deutsch, Y. Papakonstantinou, and Y. Xu. The NEXT logical framework for XQuery. In *VLDB*, 2004.

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

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

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

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

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

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

[19] I. Manolescu, V. Benzaken, A. Arion, and Y. Papakonstantinou. Structured materialized views for XML queries (extended version). INRIA HAL report 1233, available at hal.inria.fr, 2006.

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

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

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

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

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

[25] S. Paparizos, Y. Wu, L. Lakshmanan, and H. Jagadish. Tree logical classes for efficient evaluation of XQuery. In *SIGMOD*, 2004.

[26] R. Kaushik R. Krishnamurthy, V. Chakaravarthy and J. Naughton. Recursive XML schemas, recursive XML queries, and relational storage: XML-to-SQL query translation. In *ICDE*, 2004.

[27] 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.

[28] Uload web site. gemo.futurs.inria.fr/projects/XAM/.

[29] U. Washington's XML repository. www.cs.washington.edu/research/xml/datasets, 2004.

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

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

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