Structured Materialized Views for XML Queries

Ioana Manolescu, Veronique Benzaken, Andrei Arion, Yannis Papakonstantinou

Structured Materialized Views for XML Queries

Ioana Manolescu Véronique Benzaken Andrei Arion
Yannis Papakonstantinou
INRIA and LRI, France and UCSD, USA

May 13, 2006

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 materialised 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, which typically increases the opportunities for rewriting.

As a motivational example, where we will illustrate key concepts, requirements and contributions, consider the following XQuery:

```
for $x in document(‘‘XMark.xml’’)//item[id]
return <res> {$x/name/text()}
             for $y in $x//listitem
             return <key> {$y//keyword} </key>
       </res>
```

A simplified XMark document fragment appears at left in Figure 1. For conciseness, we abridge element names to the first letter when possible; thus, i stands for *item*, d for *description*, n for *name*, p for *parlist*, l for *listitem* etc. Furthermore, we assume available the views at right in Figure 1. Each view is described by a tree pattern, whereas / denote child and // denote descendent relationships, dashed edges connect optional children to their parents, and n edge label denote nested edges. The data matching the child of a nested edge will be nested under the parent match. V_1 and V_2 are materialized over the fragment at left.

V_1 stores, for every $/r//*$ element, four information items. (1) Its identifier ID_1. We consider identifiers are simple atomic values. (2) The grouped set of content of its possible d/l descendent nodes. The content 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. In Figure 1, attribute C_2 is nested in the second view attribute A_2, reflecting the nested edge. (3) The value (text children) of its possible b (*bold*) descendents. Note the null value (denoted $\perp$) representing the missing b descendents of the i_2 element. V_2 stores, for every $/r//id$ element, its identifier ID_1 and value V_1.

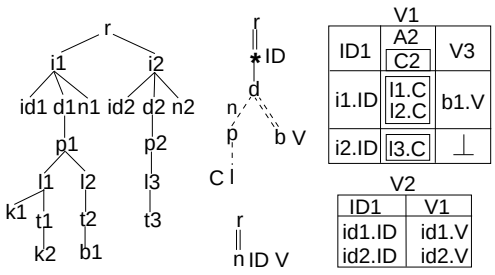

Figure 1: XMark document fragment and views materialized on this fragment.

We make the following remarks on rewriting.

(1) Although V_1 does not explicitly store data from i nodes, it may be found useful if we can infer that r children having d children are labeled i.

(2) V_1 may store some tuples that should not contribute to the query, if there are i nodes without id children. If all i nodes are guaranteed to have id children, V_1 only stores useful data.

(3) V_1 stores l elements on the path $/r/i/d/p/l$, while the query requires all l descendents of $/r/i$. V_1's l data is sufficient for the query, if we know that $/r/i//l$ and $/r/i/d/p/l$ are the same.

(4) In V_1, l elements are optional, that is, V_1 stores data from i elements without l descendents. This fits well the query, which must indeed produce output even for such i elements. The nesting of l elements under their i ancestor

is also favorable to the query, which must output such l nodes grouped in a single *res* node. Thus, the single view V_1 may be used to rewrite *across nested FLWR blocks*.

(5) Neither V_1 nor V_2 store data from k (*keyword*) nodes. However, if we know that all $/r//i//k$ nodes are descendents of some $/r//i/d/p/l$, we can extract the k elements by navigating inside the content of l nodes, stored in the $A_2.C_2$ attribute of V_1.

(6) Data from *id* 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 view definitions describe some interesting properties of the identifiers stored in the views, 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, 21, 25]. Assuming structural IDs are used, V_1 and V_2 can be combined by a *structural join* [1] on their attributes $V_1.ID_1$ and $V_2.ID_1$. Furthermore, some ID schemes also allow inferring an element's ID from the ID of one of its children [21, 25]. Assuming V_1 stored the ID of p nodes, we could derive from it the ID of their parent d nodes, and use it in other rewritings.

The reasoning in (1)-(3) and (5) 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 [16]. While the XMark DTD [29] is sufficient for (1), (2) and (5), this is not the case for the path constraint in (3). The reason is that p and l 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 [18]. A Dataguide is more precise, as it only accounts for the paths occuring in the data; it also offers some protection against a lax DTD which "hides" interesting data regularity properties.

The reasoning in (4) requires a formal materialized view model as a basis for rewriting.

Finally, the rewriting opportunities considered in (6) require: 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.

1.1 Contribution 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, described in a previous work [3]. This formalism is particularly suited for nested XQuery queries, and it extends view [6, 30] and tree pattern [2, 9, 23] formalisms used in previous works. The extraction of such patterns from an XQuery query is described in a separate work [4]. In the current work, given a query and a set of views:

- We characterize the complexity of pattern containment under Dataguide [16] and integrity constraints, and provide a containment decision algorithm.
- We describe a correct and complete view-based rewriting algorithm which combines tree pattern views into a larger pattern, equivalent (under constraints) to the target query pattern.
- The containment and rewriting algorithms have been fully implemented in the ULoad prototype, recently demonstrated [5]. We

report on their practical applicability and performance.

The novelty of our work is manifold. (*i*) Going beyond XPath views [6, 30], 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, to which we add an interesting class of integrity constraints. Dataguides can be built and maintained at low costs [16]; we show that in most practical applications, they are very compact, and can be efficiently exploited.

This paper is organized as follows. Section 2 reviews some preliminary useful notions. For the sake of readability, the presentation of the containment and rewriting issues at the heart of this paper is done in two steps. Section 3 discusses containment and rewriting for a very simple flavor of conjunctive patterns and constraints, while Section 4 gradually extends these results to the full tree pattern language and to richer constraints. Section 5 evaluates the performance of our algorithms. 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 paranthesized notation based on node labels and ignoring node IDs, e.g. $a(b\ c(d))$.

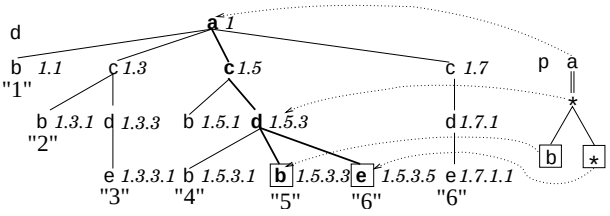

Figure 2: Sample XML document d, conjunctive pattern p, and embedding $e : p \to d$.

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, 19]. 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_2 is a /-child of n_1, $e(n_2)$ is a child of $e(n_1)$.

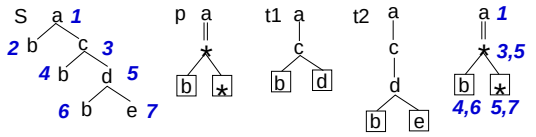

Figure 3: Summary S of the document in Figure 2, pattern p, $mod_S(p) = \{t_1, t_2\}$, and annotated p.

- For any $n_1, n_2 \in nodes(p)$ such that n_2 is a //-child of n_1, $e(n_2)$ is a descendent of $e(n_1)$.

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 $/l_1/l_2/\ldots/l_k$, $k \leq 1$, 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 can be found in the Appendix. 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 is delegated to the Appendix. 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, 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

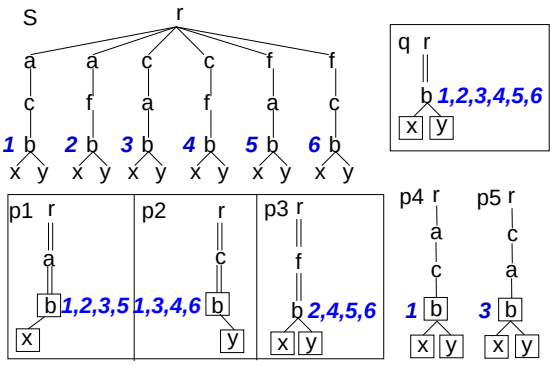

Figure 4: Summary S, query q and patterns p_1-p_5.

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.*

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 [17] 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 ?

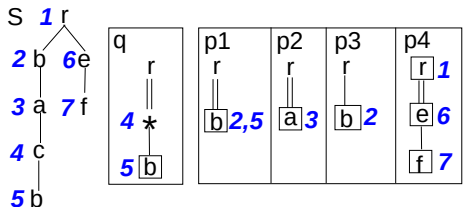

Figure 5: Sample configuration for pattern joins.

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.

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.*

Algorithm 1: Conjunctive pattern rewriting under summary constraints

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

```
1  M0 ← {(pi, pi) | 1 ≤ i ≤ n}; M ← M0
2  repeat
3      foreach (li, pi) ∈ M, (lj, pj) ∈ M0 do
4          foreach possible way of joining li
           and lj using ⋈=id, ⋈≺, ⋈≺≺ do
5              (l, p) ← (li, pi) ⋈ (lj, pj)
6              if p ≠ pi and p ≠ pj then
7                  if p ≡S q then
8                      output l
9                  else
10                     if |l| ≤ |q| × |S| then
11                         M ← M ∪ {(l, p)}
12 until M is stationary
13 foreach minimal N ⊆ M s.t.
   ∪(l,p)∈N p ≡S q do
14     output ∪(l,p)∈N p
```

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

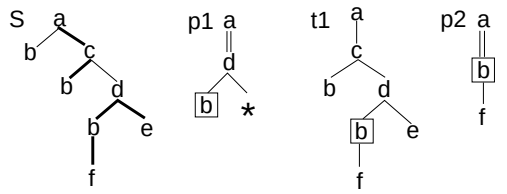

Figure 6: Enhanced summary and sample patterns.

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 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 $e : p \rightarrow S$, $mod_S(p)$ 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. We delegate its details and a full example to the Appendix.

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.

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 $e : nodes(p) \rightarrow nodes(t) \cup \{\perp\}$ 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 $e(n_2) = \perp$.*

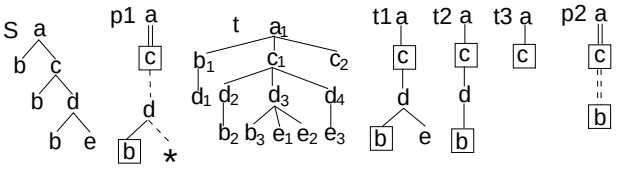

Figure 7: Optional patterns example.

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}.

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:*

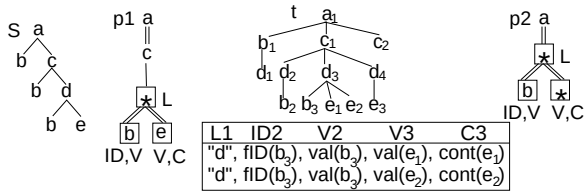

Figure 8: Attribute patterns.

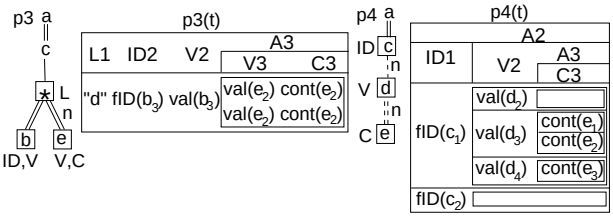

Figure 9: Nested patterns and their semantics.

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 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 $e : p_{n,1} \rightarrow S$, 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 $e' : p_{n,2} \rightarrow S$, the nesting sequence $ns(n_i^2, e')$ is similarly defined.

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 $e : p_{n,1} \rightarrow S$, there exists an embedding $e' : p_{n,2} \rightarrow S$ with the same return nodes as e, such that $ns(n_i^1, e) = ns(n_i^2, e')$.

[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.

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 [21] (illustrated in Figure 2) or Dewey IDs [25], $f_{ID}(n)$ 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, 9].

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 correspondong 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.

5 Experimental evaluation

We implemented our approach in the ULoad prototype [5], developed in Java. We focus here only on its containment and rewriting modules; the platform is described in more details in [26]. Our measures were performed on a DELL Precision M70 laptop with a 2 MHz CPU and 1 GB RAM, running Linux Gentoo, and using JDK 1.5.0 from SUN. We use XMark [29] benchmark documents (denoting by XMark$n$ a generated XMark document of n MB), and other data sets available at [27]. *All documents, patterns and summaries used are available at [26].*

Doc.	Shakespeare	Nasa	SwissProt	XMark11	XMark111	XMark233	DBLP '02	DBLP '05
Size	7.5 MB	24 MB	109 MB	11 MB	111 MB	233 Mb	133 MB	280 MB
$\|S\|$	58	24	117	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.

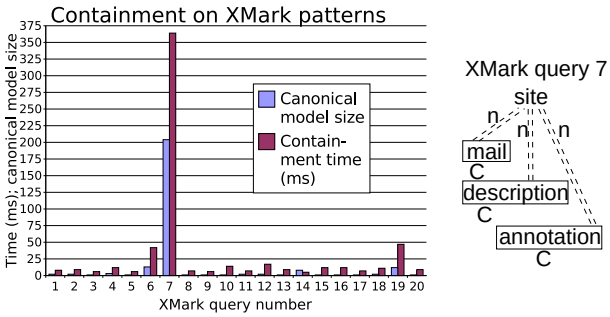

Figure 10: Containment of XMark query patterns.

5.1 Containment

We start by studying XML summaries, to see whether they can be efficiently managed. Table 1 shows the number of nodes in the summaries of several documents, including two snapshots of the DBLP database, from 2002, respectively, end of 2005. All summaries have very moderate size. The last line shows the number of strong nodes n_s, and number of one-to-one nodes n_1 in each summary. Strong and one-to-one edges are quite frequent, thus many integrity constraints can be exploited by rewriting. Table 1 also demonstrates small summary changes as the document grows: from XMark11 to XMark232, the summary only grows by a modest 10%, and similarly for the DBLP data. Intuitively, the complexity of a given data set levels off at some point. Thus, while summaries may have to be updated (in linear time [16]) with data changes, the updates are likely to be rare.

To assess the efficiency of containment, we first extracted the patterns corresponding to the 20 XMark [29] queries; the patterns have between 4 and 15 nodes. We developed their canonical models against the largest XMark summary (548 nodes), and tested the containment of each patern in itself. Figure 10 shows the canonical model size, and containment test time in milliseconds. A first remark is that $|mod_S(p)|$ is quite small, much less than the theoretical bound of $|S|^{|p|}$. Query 7 stands out, with 204 trees in its S-model. As Figure 10 shows, this is due to the lack of structural relationships between the query variables, which we expect is not the frequent case in practice. The optional edges also lead to adding trees to $mod_S(p)$, but their effect is less important (15 other XMark patterns have optional edges, yet their canonical models are small). The second remark is that the containment time is closely correlated with $|mod_S(p)|$.

Going beyond the small XMark sample, we generated synthetic, satisfiable patterns based on the 548-nodes XMark summary. Patterns have 3 to 13 nodes. with a fanout $f = 3$. Nodes were labeled $*$ with a probability $p_* = 0.1$, and with a value predicate of the form $v = c$ with probability $p_v = 0.2$. We used 10 different values. Edges were labeled // with probability $p_{//} = 0.5$ (otherwise the edge is labeled /), and were optional with probability $p_o = 0.5$. For this

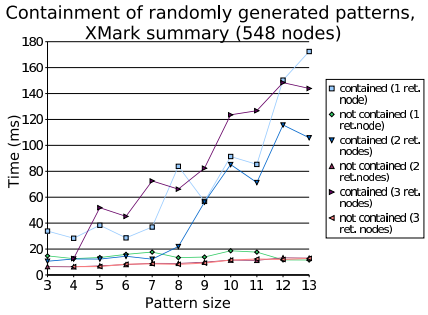

Figure 11: Containment of synthetic XMark patterns.

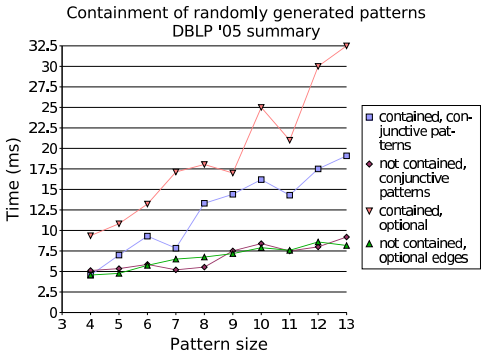

Figure 12: Containment of syntethic DBLP patterns.

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 (recall 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 totally different nodes (thus, not contained). For every n, every r, and every $i = 1, \ldots, 40$, we tested whether $p_{n,i,r}$ was included in $p_{n,j,r}$ with $j = i, \ldots, 40$, and averaged the containment time over 780 executions. Figure 11 shows the average times, separating the successful tests from the failed ones.

Note the difference between the time to perform successful containment tests, and the time for unsuccessful ones. This is because the algorithm testing $p \subseteq_S p'$ exits as soon as a tree in $mod_S(p)$ does not meet the containment criterium, which usually means $mod_S(p)$ is far from fully constructed. This is convenient to avoid spending time on unsuccessful tests. When containment holds, however, $mod_S(p)$ is fully produced and tested, thus the execution time is longer. Overall, successful tests grow longer as n grows, but the average time remain moderate. The curves are quite irregular, since $|mod_S(p)|$ varies a lot among patterns, and is difficult to control.

The times recorded in Figure 11 should be understood in correlation with the generated patterns, which are quite complex. For instance, for patterns above 6 nodes, 60% of the nodes have labels such as *emph*, *listitem*, *bold*, *keyword* etc., since these tags appear many times in the XMark summary. However, it is difficult to come up with a meaningful query referring to so many relatively low-importance elements; their labels, together, are only about 5% of the XMark query pattern nodes. To get a second glimpse at the problem, we repeated the pattern generation with the same parameters, but on the DBLP summary, with *author*, *title* and *year* as return nodes. Figure 12 shows that the time for successful containment checks is 4 times lower than for XMark, since tags such as XMark's *bold*, *emph* etc. are not present. We believe the DBLP patterns are more representative of real-life queries than the XMark synthetic ones.

Figure 12 also illustrates the impact of optional edges: the upper curves correspond to 50% of optional edges, while the lower curve cor-

responds to 0% optional edges (conjunctive patterns). The difference is approximately of a factor of 2. This confirms the impact of optional edges on containment complexity, however the impact is much more moderate than the exponential worst case (Section 4.3), demonstrating its practical applicability for patterns with optional edges.

5.2 Rewriting

We report here on the performance of our summary-based query rewriting algorithm.

We considered the query patterns extracted from the XMark [29] queries as the target patterns. The view pattern set is initialized with 2-nodes views, one per tag in the XMark document, each of them 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 view patterns based on the 548 nodes XMark summary, each of which has 3 nodes, with 50% optional edges, such that a node stores a (*structural*) ID and V with a probability $p = 0.75$. No value predicates were added.

Figure 13 shows the times to rewrite the XMark query patterns (three query patterns correspond to query 9, since the query performs a value join over three independent tree patterns [4]). For each query, we show: the time to prepare the rewriting and prune the views (applying Proposition 3.4 from Section 3), denoted *Setup and pruning time*; the time elapsed until the first equivalent rewriting is found, denoted *Time to 1st solution* (this includes the setup time); and the total rewriting time, denoted *Total time*. All times are in milliseconds.

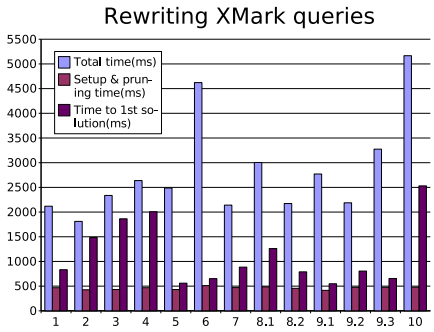

Figure 13: Performance of XMark query rewriting

The rewriting algorithm explores rewriting plans in a depth-first manner; thus, the first plans are found quite fast. This is useful since in the presence of many rewritings, a solution must be found fast, even if the search is incomplete.

The view pruning was overall very efficient: of the 183 initial views, only 8 to 20 views were retained as relevant.

Experiment conclusions Pattern containment complexity is strongly correlated with 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 found; a first rewriting is identified fast, which makes it attractive even if the rewriting is stopped before completion.

6 Related works

Containment and rewriting for semistructured queries have received significant attention in the literature, either in the general case [15, 22, 19], or under schema and other semantic con-

straints [12, 13, 20, 28]. We studied tree pattern containment in the presence of Dataguide [16] constraints which, to the best of our knowledge, had not been previously addressed. Our containment decision algorithm is related to the one presented in [19], modified to take advantage of summary constraints for extra rewritings. Summary constraints are closely related to path constraints [7], 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. The difference between schema and summary constraints is similar: a summary limits tree depth, while a (recursive) schema does not. In practical documents, recursion is present, but not very deep [18], making summaries an interesting rewriting tool. In the absence of recursion, summaries and schemas coincide are very similar. An algebraic framework for unconstrained XQuery minimization is described in [11]; we plan to study the integration of the two approaches in the future. Containment of nested XQueries has been studied in [14], based on a model without node identity, unlike our model.

Recent works have addressed materialized view-based XML query rewriting [8, 6, 30, 10]. 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 [24]. However, our approach does not need SQL as an intermediary language.

The tree patterns we consider correspond to from conjunctive nested XQuery queries, as first noted in [9, 23]. Our patterns are oriented towards modeling stored views; we formally described them in [3].

7 Conclusion

We studied the problem of XML query pattern rewriting based on summary constraints, and furthermore 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] P. Buneman, W. Fan, and S. Weinstein. Interaction between path and type constraints. *ACM Trans. Comput. Log*, 4(4), 2003.

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

[9] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[25] 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.

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

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

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

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

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

8 Appendix

The content provided here is not part of the paper, and should not be considered in its page count. It is only provided for the curiosity of the interested reader.

8.1 Prooofs of propositions 2.1 and 3.1

Proof of Proposition 2.1:
$\Leftarrow$: Let $e : p \to S$ be the embedding associated by definition to t_e. 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 $0 \leq i \leq k$. Let t_e be the tree in $mod_S(p)$ obtained from the embedding e: by choice of e, the return nodes of t_e are $(n_1^S, \ldots, n_k^S)$, where n_i^S is the path of node n_i^t for every i. Given that t_e is the minimal subtree of S to include $(n_1^S, \ldots, n_k^S)$, and since t is a tree containing nodes on all these paths, t_e is a subtree of t.

Proof of Proposition 3.1:
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:

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

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 2.1, 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 2.1) $(n_1^t, \ldots, n_k^t) \in p'(t)$.
(2) $\Leftrightarrow$ (3) follows directly from Proposition 2.1.

8.2 Containment on patterns endowed with value predicates

In Figure 14, $p_{\phi 1} - p_{\phi 4}$ 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 $\phi_1(v), \phi_2(v)$, $\neg\phi_1(v)$, $\phi_1 \vee \phi_2$, $\phi_1 \wedge \phi_2$, and $\phi_1(v) \Rightarrow \phi_2(v)$ are easily computed.

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 regular 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 14 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 14, $mod_S(p_{\phi 1}) = \{t_{\phi 1}\}$, $mod_S(p_{\phi 2}) = \{t'_{\phi 2}, t''_{\phi 2}\}$, $mod_S(p_{\phi 3}) = \{t_{\phi 3}\}$ and $mod_S(p_{\phi 4}) = \{t_{\phi 4}\}$.

Note that two pattern nodes n_x, n_y, decorated with different (or even contradictory) formulas $\phi_x(v)$, $\phi_y(v)$ may be mapped by an embedding e to the same summary node s. In this case, the canonical model tree corresponding to e must contain different nodes for $e(n_x)$ and $e(n_y)$, each labeled with its respective formula. Thus, canonical model trees in the presence of predicates are no longer strictly speaking S subtrees. However, their size remains moderate, since the p subtrees corresponding to n_x, respectively, n_y will each be reflected once in the canonical tree, under $e(n_x)$, respectively, $e(n_y)$. For simplicity, we will continue to assume here that canonical model trees are S subtrees.

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 2.1, considering decorated patterns and trees.

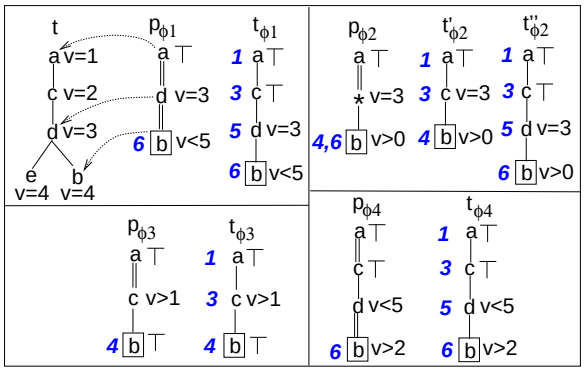

Figure 14: Decorated patterns $p_{\phi 1}, p_{\phi 2}, p_{\phi 3}$ and $p_{\phi 4}$, their canonical models, and a decorated embedding.

A characterization of S-containment among decorated patterns can be similarly obtained from Proposition 3.1. Considering two decorated patterns p_ϕ, p'_ϕ and a summary S, condition 3 from Proposition 3.1 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 14, $p_{\phi 1} \subseteq_S p_{\phi 2}$.

Characterizing the situations where $p_\phi \subseteq_S p_{\phi 1} \cup \ldots \cup p_{\phi n}$ requires some auxiliary notations. Let t_e be a tree from the S-canonical model of some decorated pattern. We denote the nodes of t_e (which are also S nodes) by $s_{i_1}, \ldots, s_{i_m}$, where $1 \leq i_1, \ldots, i_m \leq |S|$, and m is the size of t_e. For instance ,the nodes of $t_{\phi 1}$ in Figure 14 can be identified by s_1, s_3, s_5 and s_6, given the S node numbers in Figure 3. We denote by $\phi_{t_e}(v_1, \ldots, v_{|S|})$ the conjunction of the formulas attached to all t_e nodes, with the convention that each v_{i_j} is the variable corresponding to the node

s_{i_j}:

$$\phi_{t_e}(v_1, \ldots, v_{|S|}) = \phi_{s_{i_1}}(v_{i_1}) \wedge \ldots \wedge \phi_{s_{i_m}}(v_{i_m}) \mid nodes(t_e) = \{s_{i_1}, \ldots, s_{i_m}\}$$

For instance, $\phi_{t_{\phi 1}}(v_1, v_2, v_3, v_4, v_5, v_6, v_7)$ in Figure 14 is: $(v_3 = 3) \wedge (v_6 < 5)$.

We have $p_\phi \subseteq_S p_{\phi 1} \cup \ldots \cup p_{\phi n}$ iff:

1. For every $t_e \in mod_S(p_\phi)$ such that $(n_1, \ldots, n_k)$ are the return nodes of t_e, there exists some i, $1 \leq i \leq n$, such that $(n_1, \ldots, n_k) \in p_{\phi i}(t_e)$.

2. For every $t_e \in mod_S(p_\phi)$, let $f(t_e)$ be the set of patterns $p_{\phi i}$ which make condition 1. true. Let $g(t_e)$ be the set of trees from $mod_S(p)$, with $p \in f(t_e)$, having the same return nodes as t_e. Then:

$$\phi_{t_e}(v_1, \ldots, v_{|S|}) \Rightarrow \bigvee_{t'_e \in g(t_e)} (\phi_{t'_e}(v_1, \ldots, v_{|S|}))$$

Intuitively, condition 2 ensures that the value conditions attached to the nodes of p_ϕ are stricter than the disjunction of the $p_{\phi 1}, \ldots, p_{\phi m}$ conditions. The complexity of condition 2 is $N^{|S|}$, where N is the number of constants used in value comparison. In practice, we expect N to be small, moreover, the formulas typically carry over much less than S variables (as in the above example). Restricting the value predicates to equalities (drastically) reduces the complexity.

We illustrate this criterium by deciding whether $p_{\phi_2} \subseteq_S p_{\phi 1} \cup p_{\phi 3} \cup p_{\phi 4}$, for the patterns in Figure 14. We have $mod_S(p_2) = \{t'_{\phi 2}, t''_{\phi 2}\}$. We obtain (omitting the variables $(v_1, \ldots, v_7)$ from all formulas for brevity):

- $\phi_{t'_{\phi 2}}$ is $(v_3 = 3) \wedge (v_4 > 0)$, $f(t'_{\phi 2}) = \{p_{\phi 3}\}$, $g(t'_{\phi 2}) = \{t_{\phi 3}\}$, and $\phi_{t_{\phi 3}}$ is $(v_3 > 1)$. Thus, $\phi_{t'_{\phi 2}} \Rightarrow \phi_{t_{\phi 3}}$.

- $\phi_{t''_{\phi 2}}$ is $(v_5 = 3) \wedge (v_6 > 0)$, $f(t''_{\phi 2}) = \{p_{\phi 1}, p_{\phi 4}\}$, $g(t''_{\phi 2}) = \{t_{\phi 1}, t_{\phi 4}\}$, $\phi_{t_{\phi 1}}$ is $(v_5 = 3) \wedge (v_6 < 5)$, and $\phi_{t_{\phi 4}}$ is $(v_5 < 5) \wedge (v_6 > 2)$. Thus, $\phi_{t''_{\phi 2}} \Rightarrow \phi_{t_{\phi 1}} \vee \phi_{t_{\phi 4}}$.

Thus, we conclude $p_{\phi 2} \subseteq_S p_{\phi 1} \cup p_{\phi 3} \cup p_{\phi 4}$.