XPath Rewriting Using Multiple Views: Achieving Completeness and Efficiency

Bogdan Cautis*
Telecom ParisTech - LTCI
cautis@telecom-paristech.fr

Alin Deutsch†
UC San Diego
deutsch@cs.ucsd.edu

Nicola Onose
UC San Diego
nicola@cs.ucsd.edu

ABSTRACT

The standard approach for optimization of XPath queries by rewriting using views techniques consists in navigating inside a view's output, thus allowing the usage of only one view in the rewritten query. Algorithms for richer classes of XPath rewritings, using intersection or joins on node identifiers, have been proposed, but they either lack completeness guarantees, or require additional information about the data. We identify the tightest restrictions under which an XPath can be rewritten in polynomial time using an intersection of views and propose an algorithm that works for any documents or type of identifiers. As an additional contribution, we analyze the complexity of the related problem of deciding if an XPath with intersection can be equivalently rewritten as one without intersection or union.

1. INTRODUCTION

The problem of equivalently rewriting queries using views is fundamental to several classical data management tasks. While the rewriting problem has been well studied for the relational data model, its XML counterpart is not yet equally well understood, even for basic XML query languages such as XPath, due to the novel challenges raised by the features of the XML data model.

XPath [12] is the standard for navigational queries over XML data and it is widely used, either directly, or as part of more complex languages (such as XQuery [7]). Early research [24, 18, 21, 25] studied the problem of equivalently rewriting an XPath by navigating inside a *single* materialized XPath view. This is the only kind of rewritings supported when the query cache can only store or can only obtain *copies* of the XML elements in the query answer, and so the original node identities are lost.

We have recently witnessed an industrial trend towards enhancing XPath queries with the ability to expose node identifiers and exploit them using intersection of node sets (via identity-based equality). This trend is supported by such systems as [3] and has culminated in the new XPath 2.0 standard [6], which adds intersection as a first-class primitive. This development enables for the first time multiple-view rewritings obtained by intersecting several materialized view results. The single-view rewritings considered in early XPath research have only limited benefit, as many queries with no single-view rewriting can be rewritten using multiple views.

Our work is the first to characterize the complexity of the intersection-aware rewriting problem. We identify tight restrictions under which sound and complete rewriting can be performed efficiently, i.e. in polynomial time, and beyond which the problem becomes intractable (coNP hard). These restrictions are practically interesting as they permit expressive queries and views with descendant navigation and path filter predicates.

As a side-effect of our study of rewriting, we analyze the complexity of the problem of deciding if an XPath with intersection can be equivalently rewritten as one without intersection or union, case in which we say it is *union-free*. We also study the effect of intersection on the complexity of containment of XPath 2.0 queries.

Prior work on XPath containment derived coNP lower bounds in the presence of wildcard child navigation, yet showed PTIME for tree patterns without wildcard [19]. In contrast, we show that extending wildcard-free tree patterns with intersection already leads to intractability.

Running Example. Throughout the paper we will consider an example based on XPath queries over a digital library, which consists in a large number of publications, including scientific papers. A paper is organized into a hierarchy of sections, which may include, among other things, figures and images, usually related to the theorems and other results stated in the papers.

Let us assume that there has already been a query v_1, that retrieved all images appearing in sections with theorem statements:

v_1 : doc("L")//paper//section[theorem]//image

The result of v_1 is stored in the cache as a materialized view, rooted at an element named v_1. Later, the query processor had to answer another XPath v_2 looking for images inside (floating) figures that can be referenced:

v_2 : doc("L")/lib/paper//section//figure[caption//label]/image

The result of v_2 is not contained in that of v_1, so it was also executed and its answer cached.

Let us first look at an incoming query q_1, asking for all postscript images that appear in sections with theorems:

q_1 : doc("L")//paper//section[theorem]//image[ps]

q_1 can be easily answered by navigating inside the view v_1, using the following XPath query:

r_1 : doc("v_1")/v_1/image[ps]

Now, consider a query q_2 looking for the files corresponding to images inside labeled figures from sections stating theorems:

q_2 : doc("L")/lib/paper//section[theorem]//figure[caption//label]/image/file

*Part of this work was done while this author was a graduate student at INRIA Futurs. Research partially supported by the ANR grant DocFlow.

†Partially funded by NSF grants IIS-0705589, IIS-0415257 and IIS-0347968 and an Alfred P. Sloan fellowship.

Proceedings of the 11th International Workshop on Web and Databases (WebDB 2008), June 13, 2008, Vancouver, Canada

It is easy to see that q_2 cannot be answered in isolation using only v_1 or only v_2, because, for instance, there is no way to enforce that an image is both in a section having theorems and inside a labeled figure. However, by intersecting the results of the two views (assuming they both preserve the identities of the original image elements), one can build a rewriting equivalent to q_2:

$$r_2 : (\text{doc}(\text{“}v_1\text{”})/v_1/\text{image} \cap \text{doc}(\text{“}v_2\text{”})/v_2/\text{image})/\text{file}$$

Outline. The rest of the paper is organized as follows. We discuss related work in Section 2. Section 3 introduces general notions and the rewriting problem. Section 4 presents our solution and we conclude in Section 5. The proofs are in [10], Appendix E.

2. RELATED WORK

XPath rewriting using only one view (no intersection) was the target of several studies [24, 18, 21, 25]. Previously proposed join-based rewriting methods either give no completeness guarantees [3, 22] or can do so only if the query engine has extra knowledge about the structure and nesting depth of the XML document [2]. Others [22] can only be used if the node ids are in a special encoding, containing structural information. Our algorithm works for any documents and type of identifiers, including application level ids, such as the id attributes defined in the XML standard [8].

In [17] and [14], the authors look at a different problem, that of finding maximally contained rewritings of XPath queries using views. Rewriting more expressive XML queries using views was studied in [11, 13, 20], but without considering intersection.

Containment and satisfiability for several extensions of XPath with intersection have been previously investigated, but all considered problems were at least NP-hard or coNP-hard. For our language, containment is also intractable, but the equivalence test used in the rewriting algorithm is in PTIME for practically relevant restrictions. Satisfiability of XPath in the presence of the intersect operator and of wildcards was analyzed in [16], which proved its NP-completeness. As noticed in [4], there is a tight relationship between satisfiability and containment for languages that can express unsatisfiable queries. If containment is in the class K, satisfiability is in coK and if satisfiability is K-hard, containment is coK-hard. We give even stronger coNP completeness results for the containment of an XPath p_1 into an XPath p_2, by allowing intersection only in p_1 and disallowing wildcards. Satisfiability is analyzed in [4] for various fragments of XPath, including negation and disjunction, which could together simulate intersection, but lead to coPSPACE-hardness for checking containment. Richer sublanguages of XPath 2.0, including path intersection and equality, are considered in [23], where complexity of checking containment goes up to EXPTIME or higher. None of these studies tries to identify an efficient test for using intersection in query rewriting. A different approach, taken by [15] is to replace intersection by using a rich set of language features, and then try to simplify the expression using heuristics.

Finally, closure under intersection was analyzed in [5] for various XPath fragments, all of which use wildcard. We study the case without wildcard and prove that *union-freedom* (equivalence between an intersection of XPaths and an XPath without intersection or union) is coNP-hard. However, under restrictions similar to those for the rewriting problem, union-freedom can be solved in polynomial time. Thus, we also answer a question we previously raised in [9] regarding whether an intersection of XPath queries without wildcard can be reduced in PTIME to only one XPath.

3. PRELIMINARIES

We consider an XML document as an unranked, unordered rooted tree t modeled by a set of edges $\text{EDGES}(t)$, a set of nodes $\text{NODES}(t)$, a distinguished root node $\text{ROOT}(t)$ and a labeling function λ_t, assigning to each node a label from an infinite alphabet Σ.

We consider XPath queries with child / and descendant // navigation, without wildcards. We call the resulting language *XP* , and define its grammar as:

$$\begin{array}{rcl} apath & ::= & doc(\text{“}name\text{”})/rpath \mid doc(\text{“}name\text{”})//rpath \\ rpath & ::= & step \mid rpath/rpath \mid rpath//rpath \\ step & ::= & label\ pred \\ pred & ::= & \epsilon \mid [rpath] \mid [.//rpath] \mid pred\ pred \end{array}$$

The sub-expressions inside brackets are called *predicates*. As we show in [10], all definitions and results extend naturally when allowing equality with constants in the predicates.

In the following, we will prefer an alternative representation widely used in literature, the unary *tree patterns* [19]:

DEFINITION 3.1. *A* tree pattern p *is a non empty rooted tree, with a set of nodes* $\text{NODES}(p)$ *labeled with symbols from* Σ*, a distinguished node called the* output node $\text{OUT}(p)$*, and two types of edges:* child edges*, labeled by* / *and* descendant edges*, labeled by* //*. The root of* p *is denoted* $\text{ROOT}(p.)$

Any *XP* expression can be translated into a tree pattern query and vice versa (see, for instance [19]). For a given *XP* expression q, by *pattern*(q) we denote the associated tree pattern p and by *xpath*(p) $\equiv q$ the reverse transformation.

The semantics of a tree pattern can be given using embeddings:

DEFINITION 3.2. *An* embedding *of a tree pattern* p *into a tree* t *over* Σ *is a function* e *from* $\text{NODES}(p)$ *to* $\text{NODES}(t)$ *that has the following properties: (1)* $e(\text{ROOT}(p)) = \text{ROOT}(t)$*; (2) for any* $n \in \text{NODES}(p)$, $\text{LABEL}(e(n)) = \text{LABEL}(n)$*; (3) for any /-edge* (n_1, n_2) *in* p, $(e(n_1), e(n_2))$ *is an edge in* t*; (4) for any //-edge* (n_1, n_2) *in* p*, there is a path from* $e(n_1)$ *to* $e(n_2)$ *in* t.

The *result* of applying a tree pattern p to an XML tree t is the set:

$$\{(\text{ROOT}(t), e(\text{OUT}(p))) \mid e \text{ is an embedding of } p \text{ into } t\}$$

We will consider in this paper the extension $XP^\cap$ of *XP* with respect to intersection, having a straightforward semantics. The grammar of $XP^\cap$ is obtained from that of *XP* by adding the rules:

$$\begin{array}{rcl} ipath & ::= & cpath \mid (cpath) \mid (cpath)/rpath \mid (cpath)//rpath \\ cpath & ::= & apath \mid apath \cap cpath \end{array}$$

By $XP^\cap$ expressions over a set of documents D we denote those that use only *apath* expressions that navigate inside the documents D. For a fragment $\mathcal{L} \subseteq XP$, by $\mathcal{L}^\cap \subseteq XP^\cap$ we denote the $XP^\cap$ expressions that use only *apath* expressions from $\mathcal{L}$.

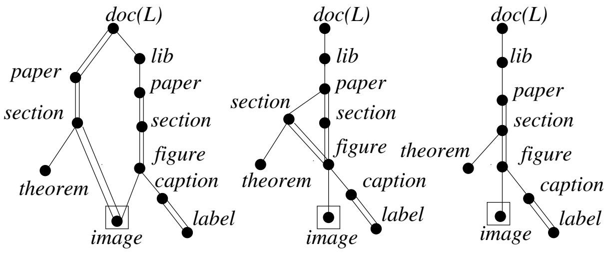
Similar to the *XP* - tree pattern duality, we can represent $XP^\cap$ expressions using the more general *DAG patterns*:

DEFINITION 3.3. *A* DAG pattern d *is a directed acyclic graph, with a set of nodes* $\text{NODES}(d)$ *labeled with symbols from* Σ*, a distinguished node called the* output node $\text{OUT}(d)$*, and two types of edges:* child edges*, labeled by* / *and* descendant edges*, labeled by* //*.* d *has to satisfy the property that any* $n \in \text{NODES}(d)$ *is accessible via a path starting from a special node* $\text{ROOT}(d)$*. In addition, all the nodes that are not on a path from* $\text{ROOT}(d)$ *to* $\text{OUT}(d)$ *(denoted* predicate nodes*) have only one incoming edge.*

Figure 1(a) gives an example of a DAG pattern. $\text{ROOT}(d)$ is the $doc(L)$ node and $\text{OUT}(d)$ is the *image* node indicated by a square.

Representing $XP^\cap$ by DAG patterns. For a query q in $XP^\cap$, we construct the associated pattern $\mathtt{dag}(q)$ as follows:

1. for every *apath* (*XP* path with no $\cap$), $\mathtt{dag}$(*apath*) is the tree pattern corresponding to the *apath* .



(a) $d = v_1 \cap v_2$ *(b) after R2.ii and R3.i* *(c) after R7*

Figure 1: Running the rules on the example of Section 1

2. $\mathtt{dag}(p_1 \cap p_2)$ is obtained from $\mathtt{dag}(p_1)$ and $\mathtt{dag}(p_2)$ as follows: (i) provided there are no labeling conflicts and both p_1 and p_2 are not empty, by coalescing $\mathrm{ROOT}(\mathtt{dag}(p_1))$ with $\mathrm{ROOT}(\mathtt{dag}(p_2))$ and $\mathrm{OUT}(\mathtt{dag}(p_1))$ with $\mathrm{OUT}(\mathtt{dag}(p_2))$ respectively, (ii) otherwise, as the empty pattern.
3. $\mathtt{dag}(x/rpath)$ and $\mathtt{dag}(x//rpath)$ are obtained as follows: (i) for non-empty x, by appending the pattern corresponding to *rpath* to $\mathrm{OUT}(\mathtt{dag}(x))$ with a /- and a //-edge respectively, (ii) as x, if x is the empty pattern.

By a pattern from language $\mathcal{L}$ we denote any pattern built as $dag(q)$, for $q \in \mathcal{L}$. Note that a tree pattern is a DAG pattern as well. The notion of *embedding* and the semantics of a pattern can be extended in straightforward manner from trees to DAGs. In the following, unless stated otherwise, all patterns are DAG patterns.

By the main branch nodes of a pattern d, $\mathrm{MBN}(d)$, we denote the set of nodes found on paths starting with $\mathrm{ROOT}(d)$ and ending with $\mathrm{OUT}(d)$. We refer to main branch paths between $\mathrm{ROOT}(d)$ and $\mathrm{OUT}(d)$ as *main branches* of d. The (unique) main branch of a tree pattern p is denoted $\mathrm{MB}(p)$. A */-pattern* is a tree pattern that has only /-edges in the main branch. We call *predicate subtree* of a pattern p any subtree of p rooted at a non-main branch node.

A *prefix* p of a tree pattern q is any tree pattern with $\mathrm{ROOT}(p) = \mathrm{ROOT}(q)$, $m = \mathrm{MB}(p)$ a subpath of $\mathrm{MB}(q)$ and having all the predicates attached to the nodes of m in q. For instance, the pattern shown in Figure 1(c) is a prefix of the pattern of q_2, since it has all the nodes of q_2, except for the output one.

DEFINITION 3.4. *A pattern d_1 is* contained *in another pattern d_2 iff for any input tree t, $d_1(t) \subseteq d_2(t)$. We write this shortly as $d_1 \sqsubseteq d_2$. We say that d_1 is* equivalent *to d_2, and write $d_1 \equiv d_2$, iff $d_1(t) = d_2(t)$ for any input tree t.*

We say that a pattern p is *minimal* [1] if there is no other pattern $p' \equiv p$ having less nodes than p.

DEFINITION 3.5. *A* mapping *between two patterns d_1 and d_2 is a function $h : \mathrm{NODES}(d_1) \to \mathrm{NODES}(d_2)$ that satisfies the properties (2),(4) of an embedding (allowing the target to be a pattern) plus three others: (5) for any $n \in \mathrm{MBN}(d_1)$, $h(n) \in \mathrm{MBN}(d_2)$; (6) for any /-edge (n_1, n_2) in d_1, $(e(n_1), e(n_2))$ is a /-edge in d_2.*

A root-mapping *is a mapping that satisfies (1). A* containment mapping *is a root-mapping h such that $h(\mathrm{OUT}(d_1)) = \mathrm{OUT}(d_2)$.*

LEMMA 3.1. *If there is a containment mapping from d_1 into d_2 then $d_2 \sqsubseteq d_1$.*

We next prove that one can always reformulate a DAG pattern as a (possibly empty) union of tree patterns. As in [5], a *code* is a string of symbols from Σ, alternating with either / or //.

DEFINITION 3.6 (INTERLEAVING). *By the* interleavings *of a pattern d we denote any tree pattern p_i produced as follows:*

1. *choose a code i and a total onto function f_i that maps $\mathrm{MBN}(d)$ into Σ-positions of i such that:*
   (a) *for any $n \in \mathrm{MBN}(d), \mathrm{LABEL}(f_i(n)) = \mathrm{LABEL}(n)$*
   (b) *for any /-edge (n_1, n_2) in d, the code i is of the form $\ldots f_i(n_1)/f_i(n_2)\ldots$,*
   (c) *for any //-edge (n_1, n_2) in d, the code i is of the form $\ldots f_i(n_1)\ldots f_i(n_2)\ldots$.*
2. *build the smallest pattern p_i such that:*
   (a) *i is a code for the main branch $\mathrm{MB}(p_i)$,*
   (b) *for any $n \in \mathrm{MBN}(d)$ and its image n' in p_i (via f_i), if a predicate subtree st appears below n then a copy of st appears below n', connected by same kind of edge.*

Two nodes n_1, n_2 from $\mathrm{MBN}(d)$ are said to be collapsed *if $f_i(n_1) = f_i(n_2)$, with f_i as above. The tree patterns p_i thus obtained are called* interleavings *of d and we denote their set by* interleave(d).

We say that a pattern d is *satisfiable* if it is non-empty and the set *interleave(d)* is non-empty. By definition, there is always a containment mapping from a satisfiable pattern into each of its interleavings. Then, by Lemma 3.1, a pattern will always contain its interleavings. Similar to a result from [5], it also holds that:

LEMMA 3.2. *Any DAG pattern is equivalent to the union of its interleavings.*

For instance, one of the seven interleavings of d in Figure 1(a) is the pattern in Figure 1(c) and another one corresponds to the XPath $doc(L)$/lib/paper//paper//section[theorem]//figure[caption[.//label]]/image The following also holds:

LEMMA 3.3. *If a tree pattern is equivalent to a union of tree patterns, then it is equivalent to a member of the union.*

Note that the set of interleavings p_i of a DAG pattern p can be exponentially larger than p. Indeed, it was shown that the $XP^{\cap}$ fragment is not included in *XP* (i.e, the union of its interleavings cannot always be reduced to one *XP* query by eliminating interleavings contained in others) and that a DAG pattern may only be translatable into a union of exponentially many tree patterns (see [5]).

DEFINITION 3.7. *We say that a DAG pattern is* union-free *iff it is equivalent to a single tree pattern.*

By Lemmas 3.2 and 3.3, a satisfiable pattern is union-free iff it has an interleaving that contains all other possible interleavings.

The rewriting problem. Given a set of views $\mathcal{V}$, defined by *XP* queries over a document D, by $D_{\mathcal{V}}$ we denote the set of view documents $\{doc(\text{“}V\text{”})|V \in \mathcal{V}\}$, in which the topmost element is labeled with the view name. Given a query $r \in XP^{\cap}$ over the view documents $D_{\mathcal{V}}$, we define *unfold*(r) as the $XP^{\cap}$ query obtained by replacing in r each $doc(\text{“}V\text{”})/V$ with the definition of V.

We are now ready to describe the view-based rewriting problem. Given a query q and a finite set of views $\mathcal{V}$ over D in a language $\mathcal{L} \subseteq XP$, we look for an alternative plan r, called a *rewriting*, that can be used to answer q. We define rewritings as follows:

DEFINITION 3.8. *For a given document D, an XP query q and XP views $\mathcal{V}$ over D, a* rewrite plan *of q using $\mathcal{V}$ is a query $r \in XP^{\cap}$ over $D_{\mathcal{V}}$. If unfold$(r) \equiv q$, then we also say r is a* rewriting.

According to the definition above and the definition of $XP^{\cap}$, a rewriting r is of the form $\mathcal{I} = (\bigcap_{i,j} u_{ij})$, $\mathcal{I}/rpath$ or $\mathcal{I}//rpath$, with u_{ij} of the form $doc(\text{“}V_j\text{”})/V_j/p_i$ or $doc(\text{“}V_j\text{”})/V_j//p_i$. We say a rewriting r is *minimal* if all p_i and all *rpath* 's are minimal.

LEMMA 3.4. *A rewrite plan can be evaluated over a set of view documents $D_{\mathcal{V}}$ in polynomial time in the size of $D_{\mathcal{V}}$.*

Completeness. In the following, by saying that an algorithm is *complete for rewriting $\mathcal{L} \subseteq XP$*, we mean that it solves the rewriting problem for queries and views in $\mathcal{L}$.

4. REWRITING ALGORITHM

Our approach for testing the existence of a rewriting (algorithm REWRITE) is the following: for each rewrite plan r using views that satisfies certain conditions w.r.t the query q, we test whether its unfolding is equivalent to q. We show that the number of plans to be considered depends only on the size of (the main branch) of q, and is thus linear. Testing equivalence between the tree pattern q and a DAG pattern d corresponding to the unfolding of r will be the central task in our algorithm. As the plans/DAGs to be considered will always contain q, testing equivalence will amount to testing the opposite containment, of d into q.

However, Lemmas 3.2 and 3.3 imply that equivalence holds iff d has an interleaving p_i such that $d \equiv p_i \equiv q$. From this observation, a naïve approach for the rewrite test would be to simply compute the interleavings of *unfold*(r) (a union of interleavings), check that this union reduces by containments to one interleaving p_i (union-freedom), and that p_i is equivalent to q. We devise an algorithm for computing the interleavings and testing union-freedom that avoids the naïve approach. It is based on a set of rewrite rules R1-R8 that simulate transformation steps of d (algorithm APPLY-RULES). Each rule application will produce an equivalent pattern that is one step closer to an interleaving that contains all others, if such a one exists. This rule-based algorithm is sound and becomes a decision procedure for union-freedom under practically relevant restrictions.

APPLY-RULES is then used in the REWRITE algorithm. While the soundness of REWRITE will follow from the soundness of APPLY-RULES, we show that it is also a decision procedure.

We next detail the algorithm that rewrites q using views $\mathcal{V}$:

REWRITE$(q, \mathcal{V})$

1 *Prefs* $\leftarrow \{(p, \{(v_i, b_i)\}) \mid v_i \in \mathcal{V}, p$ a prefix of $q, b_i \in \text{MB}(p)$, $\exists$ a mapping h from $u_i =$*pattern*(v_i) into $q, h(\text{OUT}(u_i)) = b_i\}$
2 **for** $(p, W) \in$ *Prefs*
3 **do** let $\mathcal{V}' \leftarrow \{\text{compensate}(v, p, b) \mid (v, b) \in W\}$
4 let r be the *XP*$^\cap$ query $\left(\bigcap_{v_j \in \mathcal{V}'} v_j\right)$
5 let d be the DAG corresponding to *unfold*(r)
6 APPLY-RULES(d)
7 **if** $d \sqsubseteq p$
8 **then return** $\text{compensate}(r, q, \text{OUT}(p))$
9 **return fail**

APPLY-RULES(d)

1 **repeat**
2 **repeat** apply R1 to d
3 **until** no change
4 **repeat** apply R2-R8 to d, in arbitrary order
5 **until** no change
6 **until** no change

For a pattern d and node $n \in \text{MBN}(d)$, by $\text{SP}_d(n)$ we denote the subpattern rooted at n in d. The compensate function generalizes the concatenation operation from [24], by copying extra navigation from the query into the rewrite plan. For $r \in XP^\cap$ and a tree pattern p, $\text{compensate}(r, p, n)$ returns the query obtained by deleting the first symbol from $x =$ *xpath*$(\text{SP}_p(n))$ and concatenating the rest to r. For instance, the result of compensating r = a/b with x = b[c][d]/e is the concatenation of a/b and [c][d]/e, i.e. a/b[c][d]/e. At line 8, if p is q itself, compensate returns just r, because all needed navigation had already been added at 3.

We also consider two modified versions of REWRITE :

ALL-REWRITES – same code as REWRITE with the modifications: (i) replace line 2 with: $(2')$ **for** $(p, U) \in$ *Prefs* **for** $W \subseteq U$ (ii) remove line 9 and (iii) continue to run even when the return at line 8 is reached.

EFFICIENT-RW – same code as REWRITE , except line 7, which becomes: $(7')$ **if** d is a tree **then if** $d \sqsubseteq p$.

We mention that at line 3 in the code, some elements of $\mathcal{V}'$ may be redundant and can be discarded. For space reasons, we do not discuss such optimizations.

4.1 The Rewrite Rules.

We present the rules R1-R8 as pairs formed by a test condition, which checks if the rule is applicable, and a graphical description, which shows how the rule transforms the DAG. The left-hand side of the rule description will match main branch nodes and paths in the DAG. If the matching nodes and paths verify the test conditions, then the consequent transformation is applied on them. Each transformation either (i) collapses two main branch nodes n_1, n_2 into a new node $n_{1,2}$ (which inherits the predicate subtrees, incoming and outgoing main branch edges), (ii) removes some redundant main branch nodes and edges, or (iii) appends a new predicate subtree below an existing main branch node.

Notation. We use the following notation in the graphical illustration of our rewrite rules: linear paths corresponding to part of a main branch are designated in italic by the letter p, nodes are designated by the letter n, the result of collapsing two nodes n_i, n_j will be denoted $n_{i,j}$, simple lines represent /-edges, double lines represent //-edges, simple dotted lines represent /-paths, and double dotted lines represent arbitrary paths (may have both / and //). We only represent main branch nodes or paths in the graphical description of rules (predicates are omitted). An exception is rule R5, where we refer to a subtree predicate by its *XP* expression $[Q]$. We refer to the tree pattern containing just a main branch path p simply by p, and to the tree pattern having p as main branch by $\text{TP}_d(p)$. We represent by a rhombus main branch paths that are not followed by any / (main branch) edge. Paths include their end points.

Test Conditions. In the test conditions, we say that a pattern d is *immediately unsatisfiable* if by applying to saturation Rule R1 on it we reach a pattern in which either there are two /-paths of different lengths but with the same start and end node, or there is a node with two incoming /-edges λ_1/λ and λ_2/λ, such that $\lambda_1 \neq \lambda_2$. Note that the test of immediate unsatisfiability is just a sufficient condition for the unsatisfiability of the entire DAG.

For a main branch path p in d, given by a sequence of nodes $(n_1, \ldots, n_k)$, we define $\text{TP}_d(p)$ as the tree pattern having p as main branch, n_1 as root and n_k as output, plus all the predicate subtrees (from d) of the nodes of p.

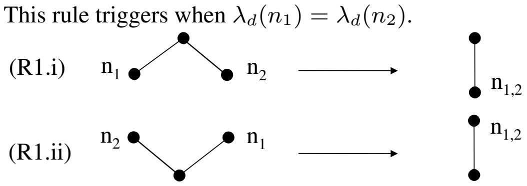
DEFINITION 4.1. *We say that two /-patterns p_1, p_2 are* similar *if (a) their main branches have the same code, and (b) both have root mappings into any /-pattern p_{12} built from p_1, p_2 as follows:*

1. *choose a code i_{12} and a total onto function f_{12} that maps the nodes of $m_{12} = \text{MBN}(p_1) \cup \text{MBN}(p_2)$ into i_{12} such that:*
   (a) *for any node n in m_{12}, $\text{LABEL}(f_{12}(n)) = \text{LABEL}(n)$*
   (b) *for any /-edge (n_1, n_2) in the main branch of p_1 or p_2, the code i_{12} contains $f_{12}(n_1)/f_{12}(n_2)$*
2. *build the minimal pattern p_{12} such that:*
   (a) *i_{12} is a code for the main branch $\text{MB}(p_{12})$,*
   (b) *for each node n in $\text{MBN}(p_1) \cup \text{MBN}(p_2)$ and its image n' in $\text{MB}(p_{12})$ (via f_{12}), if a predicate subtree st appears below n then a copy of st appears below n', connected by the same kind of edge.*

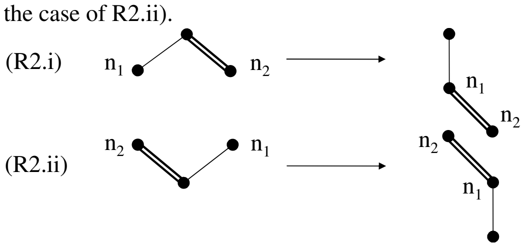
For two nodes $n_1, n_2 \in \text{MBN}(d)$, such that $\lambda_d(n_1) = \lambda_d(n_2) = \lambda$, by *collapse*$_d(n_1, n_2)$ we denote the DAG obtained from d by replacing n_1 and n_2 with a λ-labeled node $n_{1,2}$ that inherits the incoming and outgoing edges of both n_1 and n_2. We say that two nodes n_1, n_2 are *collapsible* iff they have the same label and the DAG pattern *collapse*$_d(n_1, n_2)$ is not immediately unsatisfiable.

We have now all the ingredients to present the rewrite rules:

R1 This rule triggers when $\lambda_d(n_1) = \lambda_d(n_2)$.

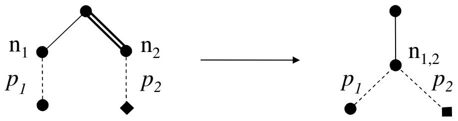


R2 This rule triggers if n_1 and n_2 are not collapsible and n_2 is not reachable from n_1 (resp. n_1 is not reachable from n_2, in the case of R2.ii).



R3 i) This rule triggers if the following conditions hold:

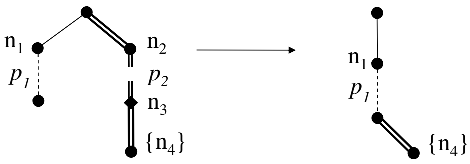
- $p_1 \equiv p_2$,
- p_2's nodes have only one incoming main branch edge,
- $\text{TP}_d(p_2)$ root-maps into $\text{TP}_d(p_1)$.



R3 ii) It is the symmetrical of R3.i) (see [10] Appendix C).

R4 i) The rule triggers if the following holds for all nodes n_4:

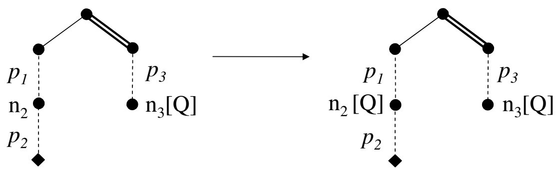
- n_3 has one incoming main branch edge, all other nodes of p_2 have one incoming and one outgoing main branch edge,
- there exists a mapping from $\text{TP}_d(p_2)$ into $\text{SP}_d(n_1)$, mapping all the nodes of p_2 into nodes of p_1.
- the path $p_2//n_4$ does not map into p_1.



R4 ii) It is the symmetrical of R4.i) (see [10] Appendix C).

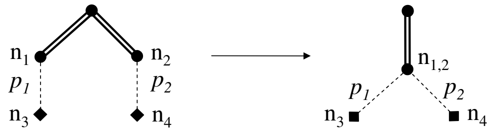
R5 This rule triggers if the following conditions hold:

- n_2 and n_3 are collapsible and $p_1 \equiv p_3$,
- *pattern*$(\lambda_d(n_2)[Q])$ has no root-mapping into $\text{SP}_d(n_2)$,
- for any node n_4 in p_2 such that $d' = \mathit{collapse}_d(n_4, n_3)$ is not immediately unsatisfiable, *pattern*$(\lambda_d(n_2)[Q])$ has a root-mapping into $\text{SP}_{d'}(n_2)$,
- if there is no path from n_3 to a node of p_2, there has to be a root-mapping from *pattern*$(\lambda_d(n_2)[Q])$ into the pattern obtained from $\text{TP}_d(p_2)$ by appending $[Q]$'s pattern, via a //-edge, below the node $\text{OUT}(\text{TP}_d(p_2))$. (Special case: p_1 and p_3 empty.)



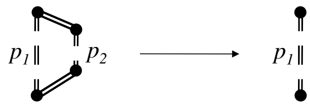
R6 This rule triggers if the following conditions hold:

- n_3, n_4 have only one incoming main branch edge, all other nodes of p_1 and p_2 have one incoming and one outgoing main branch edge,
- $\text{TP}_d(p_1)$ and $\text{TP}_d(p_2)$ are similar.

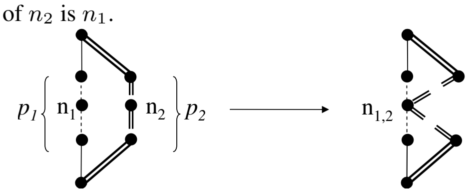


R7 This rule triggers if the following holds:

- the nodes of p_2 have only one incoming and one outgoing main branch edge,
- there exists a mapping from $\text{TP}_d(p_2)$ into $\text{TP}_d(p_1)$, such that the nodes of p_2 are mapped into nodes of p_1. (Special case: p_2 is a //-edge in parallel with p_1.)



R8 This rule triggers if in any possible mapping of p_2 into p_1 the image of n_2 is n_1.



Note that some of the rules (R3 and R6) could safely collapse more than one node, but this is done by rule R1 in any case. We opted for the current version for ease of presentation.

We illustrate in Figure 1 how we take the unfolding of the intersection of the views v_1 and v_2 from the example in Section 1 and rewrite it into a prefix of q_2 (see Figure 1.(c)). Then, line 8 in algorithm REWRITE adds the navigation /file and the rewriting r_2 that we intuitively discovered is computed.

4.2 Formal Guarantees

Using Lemma 4.1, we first show that algorithm REWRITE (and EFFICIENT-RW and ALL-REWRITES) is sound, i.e. it gives no false positives.

THEOREM 4.1. *If algorithm* REWRITE *(or* EFFICIENT-RW *or* ALL-REWRITES *) returns a DAG pattern r, then unfold*$(r) \equiv q$.

LEMMA 4.1. *The application of any of the rules from the set R1-R8 on a DAG d produces another DAG* $d' \equiv d$.

Moreover, it is also complete, in the sense described in Section 3.

THEOREM 4.2. *(1) Algorithm* REWRITE *is complete for rewriting XP . (2) If the input query q is minimal,* ALL-REWRITES *finds all minimal rewritings.*

REWRITE runs in worst-case exponential time as it uses a containment check (line 7) that is inherently hard:

THEOREM 4.3. *Containment of a query* $d \in XP^{\cap}$ *into a query* $p \in XP$ *is coNP-complete in* $|d|$ *and* $|p|$.

One might hope there is an alternative polynomial time solution. We prove this is not the case.

THEOREM 4.4. *Deciding the rewriting problem of a query q using a set of views* $\mathcal{V}$ *is coNP-complete.*

However, our rule rewriting procedure is polynomial:

LEMMA 4.2. *The rewriting of a DAG d using* APPLY-RULES *always terminates, in* $O(|\text{NODES}(d)|^2)$ *steps.*

COROLLARY 4.1. EFFICIENT-RW *always runs in PTIME.*

PTIME Completeness. We consider next restrictions by which EFFICIENT-RW becomes also complete, thus turning into a complete and efficient rewriting algorithm. Note that one may impose restrictions on either the *XP* fragment used by the query and views, or on the rewrite plans that REWRITE deals with. We consider both cases, charting a tight tractability frontier for this problem.

Case 1: *XP* fragment for PTIME. By a *//-subpredicate st* we denote a predicate subtree whose root is connected by a //-edge to a /-path p that comes from the main branch node n to which st is associated (as in $n[\ldots[.//st]]$). p is called the *incoming /-path* of st and can be empty.

By *extended skeletons* (XP_{es}) we denote patterns having the following property: for any main branch node n and //-subpredicate st of n, there is no mapping (in either direction) between the code of the incoming /-path of st and the one of the /-path following n in the main branch (where the empty code is assumed to map in any other code). Note that all the paths given in the running example are from this fragment. We can prove the following:

THEOREM 4.5. *For any pattern d in $XP_{es}^{\cap}$, d is union-free iff the algorithm* APPLY-RULES *rewrites d into a tree.*

COROLLARY 4.2 (XP_{es}). *Algorithm* EFFICIENT-RW *is complete for rewriting XP_{es}.*

Case 2: Rewrite-plans for PTIME. We also identify a large class of rewrite plans that lead to PTIME completeness. Let us first introduce the notion of */-tokens* of a tree pattern. More specifically, the main branch of a tree pattern p can be partitioned by its sub-sequences separated by //-edges, and each /-pattern from this partitioning is called a *token*. We can thus see a pattern p as a sequence of tokens (/-patterns) $p = t_1//t_2//\ldots//t_k$. We call t_1, the token starting with $\mathrm{ROOT}(p)$, the *root token* of p. The token t_k, which ends by $\mathrm{OUT}(p)$, is called the *result token* of p.

We say that two (or several) tree patterns are *akin* if their root tokens have the same main branch codes. For instance, while the views v_1 and v_2 from our example are not akin, v_1 is akin to:

v_2' : doc("L")//figure[.//caption//label]//subfigure/image[ps].

In this setting, we can relax the syntactic restrictions and accept the class of patterns $XP_{//}$, obtained from extended skeletons by freely allowing //-edges in the predicates that are connected by a //-edge to the main branch (such as in v_2'). We can prove the following:

THEOREM 4.6. *For DAGs of the form $d = \bigcap_j p_j$, where all p_j are in $XP_{//}$ and akin, d is union-free iff the algorithm* APPLY-RULES *rewrites d into a tree.*

COROLLARY 4.3 ($XP_{//}$). EFFICIENT-RW *always finds a rewriting for $XP_{//}$, provided there is at least a rewriting r such that the patterns intersected in unfold(r) are akin.*

Tractability Frontier. We show next that relaxing any of these restrictions leads to hardness for rewriting and union-freedom:

THEOREM 4.7. *(1) For a pattern d in $XP_{//}^{\cap}$, deciding if d is union-free is coNP-complete. (2) For a pattern $d = \bigcap_j p_j$, where all p_j are in XP and akin, deciding if d is union-free is coNP-hard.*

Please note that the rewriting problem can be solved using an oracle for union-freedom, but this does not provide any easy map reduction. This is why we prove the following result independently:

THEOREM 4.8. *(1) Deciding the existence of a rewriting for a query and views from $XP_{//}$ is coNP-complete. (2) For a query and views from XP, deciding the existence of a rewriting r such that the patterns intersected in unfold(r) are akin is coNP-complete.*

Discussion. We mention that all the results in this paper also apply when we add equalities with constants into the language. The extension is presented in the long version [10] (see Appendix B for definitions and Appendix C for the extended rewriting rules).

5. CONCLUSION

Our work identifies the tightest restrictions under which an XPath query can be rewritten in PTIME using an intersection of views. A side effect of this research is to establish a similar tractability frontier for the problem of deciding if an intersection of XPaths can be equivalently rewritten as an XPath without intersection or union. As future work, we plan to extend our techniques to XPath rewritings with multiple levels of intersection and union.

6. REFERENCES

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

[2] A. Arion, V. Benzaken, I. Manolescu, and Y. Papakonstantinou. Structured materialized views for XML queries. In *VLDB*, 2007.

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

[4] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. In *PODS*, 2005.

[5] M. Benedikt, W. Fan, and G. Kuper. Structural properties of XPath fragments. *Theor. Comput. Sci.*, 336(1), 2005.

[6] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon. XML path language (XPath) 2.0, 2007.

[7] S. Boag, D. Chamberlain, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language, 2007.

[8] T. Bray, J. Paoli, C. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (XML) 1.0 (fourth edition), 2006.

[9] B. Cautis, S. Abiteboul, and T. Milo. Reasoning about XML update constraints. In *PODS*, 2007.

[10] B. Cautis, A. Deutsch, and N. Onose. XPath views for documents with persistent identifiers, 2008. TR CS2008-0920, UCSD. Available from http://db.ucsd.edu/index.jsp?pageStr=publications.

[11] L. Chen and E. A. Rundensteiner. XCache: XQuery-based caching system. In *WebDB*, 2002.

[12] J. Clark and S. DeRose. XML path language (XPath), 1999.

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

[14] J. Gao, T. Wang, and D. Yang. MQTree based query rewriting over multiple XML views. In *DEXA*, 2007.

[15] S. Groppe, S. Böttcher, and J. Groppe. XPath query simplification with regard to the elimination of intersect and except operators. In *ICDE Workshops*, 2006.

[16] J. Hidders. Satisfiability of XPath expressions. In *DBPL*, 2003.

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

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

[19] G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *J. ACM*, 51(1), 2004.

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

[21] J. Tang and S. Zhou. A theoretic framework for answering XPath queries using views. In *XSym*, 2005.

[22] N. Tang, J. X. Yu, M. T. Özsu, B. Choi, and K.-F. Wong. Multiple materialized view selection for XPath query rewriting. In *ICDE*, 2008.

[23] B. ten Cate and C. Lutz. The complexity of query containment in expressive fragments of XPath 2.0. In *PODS*, 2007.

[24] W. Xu and Z. M. Özsoyoglu. Rewriting XPath queries using materialized views. In *VLDB*, 2005.

[25] L. H. Yang, M. L. Lee, and W. Hsu. Efficient mining of XML query patterns for caching. In *VLDB*, 2003.