Logic-Based XPath Optimization

Pierre Genevès
INRIA Rhône-Alpes
Zirst, 655 avenue de l'Europe, Montbonnot
38334 Saint Ismier Cedex, France
pierre.geneves@inria.fr

Jean-Yves Vion-Dury*
INRIA Rhône-Alpes / XRCE
Zirst, 655 avenue de l'Europe, Montbonnot
38334 Saint Ismier Cedex, France
jean-yves.vion-dury@xrce.xerox.com

ABSTRACT

XPath [5] was introduced by the W3C as a standard language for specifying node selection, matching conditions, and for computing values from an XML document. XPath is now used in many XML standards such as XSLT [4] and the forthcoming XQuery [10] database access language. Since efficient XML content querying is crucial for the performance of almost all XML processing architectures, a growing need for studying high performance XPath-based querying has emerged. Our approach aims at optimizing XPath performance through static analysis and syntactic transformation of XPath expressions.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Query formulation*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Performance evaluation (efficiency and effectiveness)*; D.3.4 [**Programming Languages**]: Processors—*Optimization*

General Terms

XPath, XML, query, containment, axiomatization, optimization, efficiency.

1. INTRODUCTION

In this paper, we first describe a formal architecture for static analysis that could be implemented independently from any particular XPath engine. Second, as an application, we show how the containment relation over XPath expressions can be used for general optimization purposes. The important advantage of our approach is that logic-based optimizations can be applied at syntactic level and thus remain compatible with any XPath engine. Hence, one does not have to modify an XPath engine to optimize XPath queries.

1.1 Related Work

Abundant literature on query rewriting for optimization can be found in database systems. As XML is becoming the de facto standard for representing structured content, the mapping between heritage of database theory and structured documents querying is being worked out. Optimization results on tree patterns [2, 11] or regular paths [1] often rely on a notion of query *equivalence* or query *containment* with respect to the considered model. Our approach also relies on a similar relation but aims at studying and taking XPath semantics peculiarities into account, in order to be extensible to a large XPath fragment. The containment over XPath expressions is defined using XPath semantics, usually described by a formal semantics function $\mathcal{S}$ (as found in [17]). Thus, the containment relation between two XPath expressions p_1 and p_2 holds when, for any XML tree t and any context node x of t, the set-theoretic inclusion relation holds between the sets of nodes respectively returned by the evaluation of p_1 and p_2:

$$p_1 \leq p_2 \quad \text{iff} \quad \forall t, \forall x \in t, \mathcal{S}[\![p_1]\!]_x \subseteq \mathcal{S}[\![p_2]\!]_x$$

Containment for XPath expressions is being actively studied [13, 8, 14, 18, 16], but none of these approaches explains how the containment can be practically used for XPath optimization. A rewriting approach [15] proposed to rewrite XPath backward axes into forward ones, in order to optimize XPath queries for stream-based processing. However, this rewriting technique mostly relies on XPath symmetry and does not involve containment-based optimization. We believe the present work is a first step toward the use of the containment for XPath optimization.

1.2 Outline

In section 2 we present the abstract syntax and semantics of XPath expressions we consider. The next sections detail our syntactic transformations: our global architecture is described in section 3 and we explain how the containment can be used for XPath optimization in section 4. We then describe and analyze preliminary experiments in section 5 before we conclude in section 6.

2. XPATH SYNTAX AND SEMANTICS

In this paper, we consider a subset of the XPath specification [5], mainly composed of forward axes and qualifiers. Our intent is to extend this subset to more features,

*Visiting researcher from Xerox Research Centre Europe.

DocEng'04, October 28–30, 2004, Milwaukee, Wisconsin, USA.

Path	p	::=	$\wedge \mid \bot \mid p_1 \mid p_2 \mid p_1 \cap p_2 \mid p_1/p_2 \mid (p) \mid p[q] \mid a::N$
Qualifier	q	::=	**true** $\mid$ **false** $\mid (q) \mid$ **not** $q \mid q_1$ **or** $q_2 \mid q_1$ **and** $q_2 \mid p_1 \sqsubseteq p_2$
Axis	a	::=	child $\mid$ descendant $\mid$ self $\mid$ descendant-or-self $\mid$ attribute $\mid$ namespace
NodeTest	N	::=	$n \mid * \mid$ **node**() $\mid$ **text**() $\mid$ **element**() $\mid$ **processing-instruction**() $\mid$ **comment**()

Figure 1: Considered XPath Abstract Syntax.

such as "count()" and "position()" in qualifiers. Our XPath fragment includes several variants: the void path $\bot$ and the explicit root node $\wedge$ (respectively proposed and defined in [15] and [16]) to ease formal analysis and to make the XPath syntax fully compositional. In addition, we chose to include two XPath 2.0 [3] extensions: qualified paths (e.g. $(p)[q]$) instead of qualified steps (e.g. $a::N[q]$) and path intersection ($p_1 \cap p_2$). Our fragment also includes an important extension with respect to qualifiers: the node set inclusion constraint $p_1 \sqsubseteq p_2$, defined in [16], which brings extra - yet tractable - expressive power. Note that the usual form $p_1[p_2]$ is a syntactic sugar for $p_1[\textbf{not}\ \ (p_2 \sqsubseteq \bot)]$. In the XPath abstract syntax shown below, a node test n denotes any element name: The denotational semantics of a path p provided a context node x, noted $\mathcal{S}[\![p]\!]_x$, is given in appendix A. An originality comes from the addition of the inclusion constraint between two paths inside qualifiers: $\mathcal{Q}[\![p_1 \sqsubseteq p_2]\!]_x = \mathcal{S}[\![p_1]\!]_x \subseteq \mathcal{S}[\![p_2]\!]_x$. Note that this definition allows us to define a full path equality test $p[p_1 == p_2]$ as $p[p_1 \sqsubseteq p_2$ **and** $p_2 \sqsubseteq p_1]$ whereas the standard construct $p[p_1 = p_2]$ is equivalently expressed as $p[p_1 \cap p_2]$.

3. FORMAL ARCHITECTURE FOR STATIC ANALYSIS

Formal methods for static analysis of XPath expressions have to face the combinatorial complexity of any significant fragment of XPath syntax. A way to ease analysis is to simplify the form of XPath expressions. To this end, our work currently focuses on rewriting an XPath expression into an equivalent but simplified form, using semantic-preserving rules. This normalization aims at easing whatever subsequent analysis operations. As a particular application, we show in section 4 how to use the containment relation to perform generic optimizations. Figure 2 gives an applicative view of our architecture.

3.1 Normalization Phase

More formally, a path p is rewritten into its normal form $\overline{p}$ using two sets of rules $\mathcal{N}$ and $\mathcal{C}$. This transformation is noted $p \xrightarrow{\mathcal{N}^*} \xrightarrow{\mathcal{C}^*} \overline{p}$ and first uses the set of rules $\mathcal{N}$ (see [16]) mainly composed of distributivity rules such as $(p_1 \mid p_2)/p \rightarrow p_1/p \mid p_2/p$ and other structural rules needed to reach the disjunctive normal form structure. The second step of the normalization phase involves the set of rules $\mathcal{C}$:

p[**false**]	$\rightarrow$	$\bot$	(r4f)
$\bot[q]$	$\rightarrow$	$\bot$	(r4e)
$\bot/p$	$\rightarrow$	$\bot$	(r4a)
$p/\bot$	$\rightarrow$	$\bot$	(r4b)
$\bot \mid p$	$\rightarrow$	p	(r4c)
$p \mid \bot$	$\rightarrow$	p	(r4cp)
$n_1 \neq n_2$, $a::n_1[q_1]/\text{self}::n_2[q_2]$	$\rightarrow$	$\bot$	(ra24)

that performs void path elimination: rule ra24 detects a possible contradiction; void paths are propagated to the top level of the path structure (using r4f, r4e, r4a, and r4b) and then eliminated using r4c and r4cp. At the end of the normalization phase, a normal path $\overline{p}$ is either the void path $\bot$ or a very constrained disjunction, as captured by the right graph on figure 3, which represents the grammatical structure of terms before and after the normalization. Intuitively, the set of normal paths is isomorphic to the transitive closure of the graph. A normal path also verifies additionnal properties not captured by structural constraints. For example, union subterms are syntactically different, and the root node $\wedge$ only occurs either in the first or in the second step of the expression. To prove the existence of such a normal form for any path (work in progress), we are building a logical model of the rewriting system using the calculus of inductive constructions, beyond the scope of this paper. We are currently working on the normalization issues (in particular the termination and confluence proofs; see [7] for general issues) with the Coq proof assistant [6].

3.2 Optimization Phase

The normal path $\overline{p}$ is transformed into an optimized path p' using a combination of two successive rewritings ; this is noted $\overline{p} \xrightarrow{\mathcal{O}^*} \xrightarrow{\mathcal{C}^*} p'$ or equivalently $\overline{p} \xrightarrow{\mathcal{C}^* \circ \mathcal{O}^*} p'$. Note that the rewriting $\mathcal{C}$, already used during normalization, is applied again in order to eliminate void paths potentially introduced by $\mathcal{O}$. The following section details how a path can be optimized using the containment relation over XPath expressions.

4. USING CONTAINMENT FOR XPATH OPTIMIZATION

4.1 Deciding Containment

The XPath fragment we consider in this paper is quite close to the one found in [14], for which the containment has been shown to be decidable. In order to assert containment facts, we rely on an inference and rewriting based approach described in [16]. The latter uses logical rules à la Hilbert (see [9] for a comprehensive introduction to logical formalisms):

$$\frac{A_1 \ldots A_n}{B}\ \mathrm{r}$$

where A_i and B are judgments over paths and qualifiers, and r is the rule name. Such a logical rule means that if all judgments A_i are true, then B is true. For example, the containment rule c1 states that the void path is contained in any other path, c2 constructs the reflexivity directly, d1 addresses the comparison of steps and d2 captures the general

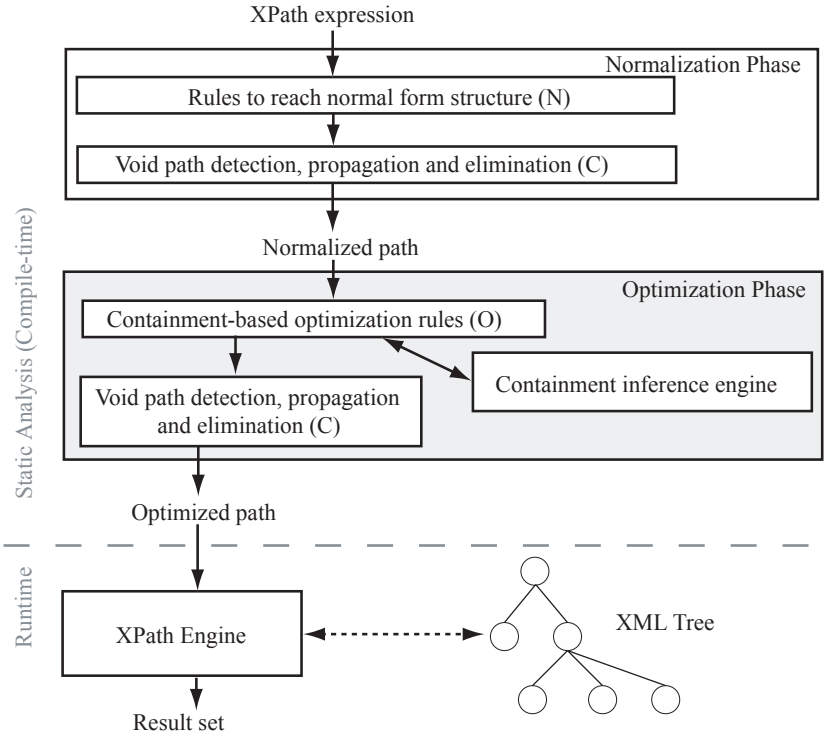

Figure 2: Applicative View of the Transformation Architecture.

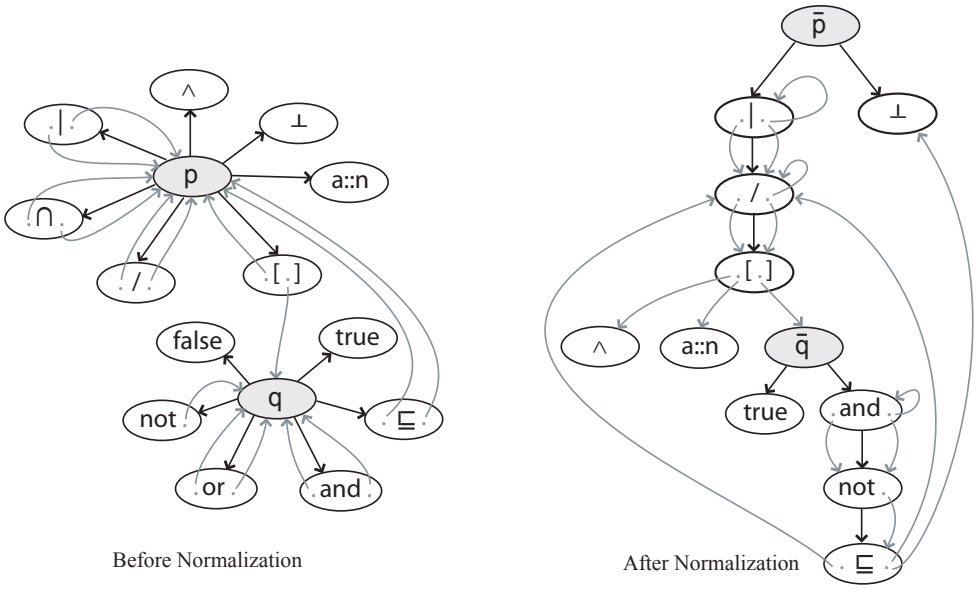

Figure 3: Stratification and Associativity of Operators.

behavior of containment relation w.r.t. the slash operator:

$$\frac{}{\bot \leq p}\ \text{c1} \qquad \frac{}{p \leq p}\ \text{c2} \qquad \frac{a_1 \leq a_2 \quad N_1 \leq N_2}{a_1{::}N_1 \leq a_2{::}N_2}\ \text{d1}$$

$$\frac{p_1[p_2] \leq p_3[p_4] \quad p_2 \leq p_4}{p_1/p_2 \leq p_3/p_4}\ \text{d2} \qquad \frac{p_1 \leq p_2 \quad q_1 \Rightarrow q_2}{p_1[q_1] \leq p_2[q_2]}\ \text{d3}$$

The comparison of steps (rule d1) relies on a partial ordering of XPath axes and nodetests:

$$\begin{gathered}\text{self} \leq \text{descendant-or-self} \\ \text{child} \leq \text{descendant} \leq \text{descendant-or-self} \\ n \leq * \leq \textbf{node}()\end{gathered}$$

The containment relation $\leq$ is also defined using a dual relation $\Rightarrow$, the logical implication between qualifiers. Hence, the comparison of qualified paths through rule d3 involves the qualifier implication, and rules e_i handles connectives:

$$\frac{p_3 \leq p_1 \quad p_2 \leq p_4}{p_1 \sqsubseteq p_2 \Rightarrow p_3 \sqsubseteq p_4}\ \text{e1} \qquad \frac{q_2 \Rightarrow q_1}{\textbf{not}\ q_1 \Rightarrow \textbf{not}\ q_2}\ \text{e2}$$

$$\frac{q_1 \Rightarrow q}{q_1\ \textbf{and}\ q_2 \Rightarrow q}\ \text{e4} \qquad \frac{q \Rightarrow q_1 \quad q \Rightarrow q_2}{q \Rightarrow q_1\ \textbf{and}\ q_2}\ \text{e5}$$

Two techniques allow us to reduce the size of the axiomatic system: the first one is an equivalence relation $\equiv$ that typically captures commutativity and associativity, and which is fully (left and right) congruent w. r. t. the containment and implication[1] , and the fundamental rule h which involves the normalization process $\mathcal{N}$ in order to transform the operands:

$$\frac{p_1 \xrightarrow{\mathcal{N}^*} p_1' \quad p_2 \xrightarrow{\mathcal{N}^*} p_2' \quad p_1' \leq p_2'}{p_1 \leq p_2}\ \text{h}$$

For more details on the approach, with a larger XPath fragment, the reader can refer to [16]. We are currently working on a full characterization of this axiomatic system[2], using the Coq proof assistant [6] and semi-automated strategies in order to tackle the combinatorial complexity. An algorithm for deciding the containment is considered as a proof tree computation.

4.2 Containment-Based Optimization Rules

Our system is basically the set $\mathcal{O}$ of conditional term rewriting rules (see figures 4,5,7,6). Numerators (top side of "fractions") are logical conditions that must be satisfied by terms or subterms in order to apply the rules. Most of the rules make use of the containment relation, and thus, each reduction step must be justified by using the inference system for containment. We distinguishes two sets of rules:

1. *redundancy elimination.* Rules ru1, ru2, ru3, ru4, ru5 of figure 4 handle union operator; rules rs1 and rs2 (figure 5) eliminate qualifier conditions induced by path composition (through the / operator), such as $a[*]/b$. This latter case is formally optimized through application of rs1 as follow

$$\cfrac{\cfrac{\cfrac{b \leq * \quad \bot \leq \bot}{* \sqsubseteq \bot \Rightarrow b \sqsubseteq \bot}\ \text{e1}}{\textbf{not}\ b \sqsubseteq \bot \Rightarrow \textbf{not}\ * \sqsubseteq \bot}\ \text{e2}}{\cfrac{\textbf{not}\ b \sqsubseteq \bot \Rightarrow *}{a[*]/b \rightarrow a/b}\ \equiv}\ \text{rs1}$$

   The rules rs3, rs4 (also figure 5) manage cases where the qualifier is naturally induced by the path, as in $a[\text{self::}*]$ and the last subset from figure 6 is dedicated to qualifier simplification.

2. *void paths elimination.* Some implicit contradictions (that result in void paths) are not captured in $\mathcal{N}$ and $\mathcal{C}$ rewriting systems, because it requires some significant inference power. Rules of figure 7 aims to detect these cases using the path containment or qualifier implication proof system.

The generic rules r-gen, r-union and r-and handle associative-commutative equivalence of terms (see 8). The $\approx$ relation is a syntactic equality up to commutativity and associativity of union, "and" and "or" operators. for instance $a[b\ \textbf{and}\ c] \approx a[c\ \textbf{and}\ b]$ because $b\ \textbf{and}\ c \approx c\ \textbf{and}\ b$
In any axiomatic system construction, the selection of axioms is based on "intuition" and therefore conveys some arbitrary decisions; then, the effort to reach consistency and completeness brings light on the underlying choices behind the axioms, and often provide rationale in order to reorganize, simplify or extend the theory. After developing a short illustration, we will propose some hints in this direction. Let us consider the following example (the "descendant" axis is abbreviated into "desc")

EXAMPLE 1. $\quad A = \wedge//a[*/b/c\ \textbf{and}\ \textit{desc::}b]$

The first normalization step expands the syntactic sugar (here $\wedge//a$). Then the desc-or-self axis is transformed into a disjunctive term and finally, qualifiers are introduced at each path step. In the notation below, $\xrightarrow{\mathcal{N}}$ denotes a one-step derivation from the rewriting system $\mathcal{N}$ and $\xrightarrow{\mathcal{N}^+}$ denotes the transitive closure of this relation.

$$\begin{aligned} A &\xrightarrow{\mathcal{N}} \wedge/\text{desc-or-self::}*/a[*/b/c\ \textbf{and}\ \text{desc::}b] \\ &\xrightarrow{\mathcal{N}^+} \wedge/\text{desc::}*/a[*/b/c\ \textbf{and}\ \text{desc::}b] \\ &\quad\ \mid\ \wedge/\text{self::}*/a[*/b/c\ \textbf{and}\ \text{desc::}b] \\ &\xrightarrow{\mathcal{N}^+} \wedge[\textbf{true}]/\text{desc::}*[\textbf{true}]/\text{child::}a[Q] \\ &\quad\ \mid\ \wedge[\textbf{true}]/\text{child::}a[Q] \quad = \overline{A} \end{aligned}$$

$$\text{with} \quad Q = \text{child::}*[\textbf{true}]/\text{child::}b[\textbf{true}]/\text{child::}c[\textbf{true}]\ \textbf{and}\ \text{desc::}b[\textbf{true}]$$

At this stage, several optimizing rewriting steps can be applied using rules from $\mathcal{O}$, described on figures 4,5,6:

$$\begin{aligned} \overline{A} &\xrightarrow{\text{rs3,rs3,rs3}} \wedge/\text{desc::}*/\text{child::}a[Q] \mid \wedge/\text{child::}a[Q] \\ &\xrightarrow{\text{ru4}} (\wedge/\text{desc::}*/\text{child::}a \mid \wedge/\text{child::}a)[Q] \\ &\xrightarrow{\text{ru5}} (\wedge \mid \wedge)/\text{desc::}a[Q] \\ &\xrightarrow{\text{ru1}} \wedge/\text{desc::}a[Q] \\ &\xrightarrow{\text{rq1,rs3}^*} \wedge/\text{desc::}a[\text{child::}*/\text{child::}b/\text{child::}c] \\ &= \wedge/\text{desc::}a[*/b/c] \end{aligned}$$

[1] left congruence: for all p_1, p_2, p if $p_1 \leq p$ and $p_1 \equiv p_2$, then $p_2 \leq p$

[2] We are currently investigating properties such as soundness and completeness.

$$\frac{p_1 \leq p_2}{p_1 \mid p_2 \rightarrow p_2}\;\text{ru1} \qquad \frac{p_1 \leq p_2}{p_1/p_3 \mid p_2/p_4 \rightarrow p_2/(p_3 \mid p_4)}\;\text{ru2} \qquad \frac{p_3 \leq p_4 \quad p_4 \leq p_3}{p_1/p_3 \mid p_2/p_4 \rightarrow (p_1 \mid p_2)/p_4}\;\text{ru3}$$

$$\frac{q_1 \Rightarrow q_2 \quad q_2 \Rightarrow q_1}{p_1[q_1] \mid p_2[q_2] \rightarrow (p_1|p_2)[q_1]}\;\text{ru4} \qquad \frac{a \in \{\text{desc}, \text{child}\}}{p_1/\text{desc}::*/\text{a}::N \mid p_2/\text{a}::N \rightarrow (p_1 \mid p_2)/\text{desc}::N}\;\text{ru5}$$

Figure 4: Containment-Based Optimization Rules ($\mathcal{O}$): union operator .

$$\frac{\textbf{not }(p_2 \sqsubseteq \bot) \Rightarrow q}{p_1[q]/p_2 \rightarrow p_1/p_2}\;\text{rs1} \qquad \frac{\textbf{not }(p_2 \sqsubseteq \bot) \Rightarrow q_1}{p_1[q_1 \textbf{ and } q_2]/p_2 \rightarrow p_1[q_2]/p_2}\;\text{rs2} \qquad \frac{\textbf{not }(p \sqsubseteq \bot) \Rightarrow q}{p[q] \rightarrow p}\;\text{rs3} \qquad \frac{\textbf{not }(p \sqsubseteq \bot) \Rightarrow q_1}{p[q_1 \textbf{ and } q_2] \rightarrow p[q_2]}\;\text{rs4}$$

Figure 5: Containment-Based Optimization Rules ($\mathcal{O}$): paths vs qualifiers.

$$\frac{q_1 \Rightarrow q_2}{q_1 \textbf{ and } q_2 \rightarrow q_1}\;\text{rq1} \qquad \frac{q_1 \Rightarrow q_2}{q_1 \textbf{ and } (\textbf{not } q_2) \rightarrow \textbf{false}}\;\text{rq2} \qquad \frac{p_1 \leq p_2}{p_1 \sqsubseteq p_2 \rightarrow \textbf{true}}\;\text{rq3} \qquad \frac{p_1 \cap p_2 \leq \bot}{p_1 \sqsubseteq p_2 \rightarrow \textbf{false}}\;\text{rq4}$$

Figure 6: Containment-Based Optimization Rules ($\mathcal{O}$): qualifier simplification.

$$\frac{q \Rightarrow (p_2 \sqsubseteq \bot)}{p_1[\textbf{not } q]/p_2 \rightarrow \bot}\;\text{rv1} \qquad \frac{q \Rightarrow (p \sqsubseteq \bot)}{p[\textbf{not } q] \rightarrow \bot}\;\text{rv2} \qquad \frac{q_1 \Rightarrow (p_2 \sqsubseteq \bot)}{p_1[(\textbf{not } q_1) \textbf{ and } q_2]/p_2 \rightarrow \bot}\;\text{rv1p} \qquad \frac{q_1 \Rightarrow (p \sqsubseteq \bot)}{p[(\textbf{not } q_1) \textbf{ and } q_2] \rightarrow \bot}\;\text{rv2p}$$

Figure 7: Containment-Based Optimization Rules ($\mathcal{O}$): contradictions.

$$\frac{p \approx p' \qquad p' \rightarrow p''}{p \rightarrow p''}\;\text{r-gen}$$

$$i < j \ \text{ or } j < i \begin{cases} \dfrac{p_i \mid p_j \rightarrow p}{p_1 \mid \cdots p_i \cdots p_j \cdots \mid p_n \rightarrow p_1 \mid \cdots p \cdots \mid p_n}\;\text{r-union} \\[2ex] \dfrac{q_i \textbf{ and } q_j \rightarrow q}{q_1 \textbf{ and } \cdots q_i \cdots q_j \cdots \textbf{ and } q_n \rightarrow q_1 \textbf{ and } \cdots q \cdots \textbf{ and } q_n}\;\text{r-and} \end{cases}$$

Figure 8: Generic Optimization Rule ($\mathcal{O}$)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{. \leq . \quad \dfrac{\text{child} \leq \text{desc} \quad * \leq *}{\text{child}::* \leq \text{desc}::*}\,\text{d1}}{./\text{child}::* \leq ./\text{desc}::*}\,\text{d2} \quad \dfrac{\text{child} \leq \text{desc} \quad b \leq b}{\text{child}::b \leq \text{desc}::b}\,\text{d1}}{*/b \leq \text{desc}::b}\,\text{g1} \quad \dfrac{}{c \Rightarrow \textbf{true}}\,\text{e3a}}{*/b[c] \leq \text{desc}::b}\,\text{d3} \quad \dfrac{}{\bot \leq \bot}\,\text{c1}}{\text{desc}::b \sqsubseteq \bot \ \Rightarrow\ */b[c] \sqsubseteq \bot}\,\text{e1}}{\text{desc}::b \sqsubseteq \bot \ \Rightarrow\ */b/c \sqsubseteq \bot}\,\text{f2}}{\textbf{not } */b/c \sqsubseteq \bot \ \Rightarrow\ \textbf{not } \text{desc}::b \sqsubseteq \bot}\,\text{e2}}{*/b/c \Rightarrow \text{desc}::b}\,\equiv}{p[*/b/c \textbf{ and } \text{desc}::b] \rightarrow p[*/b/c]}\,\text{rq1}$$

Figure 9: Proof Tree.

So what do we gain? First the descendant-or-self axis is gone, replaced by a descendant axis; second, the desc::b qualifier is gone, and this is certainly the most interesting -and difficult- point, we have to detail here after. To achieve this step, we must use the inference system, as the rule $rq1$ requires to prove that if $*/b/c$ is not empty, then so is desc::b. Figure 4.2 presents how this statement is inferred in our system.

4.3 Discussion

Through example 1, we outlined the interest of inferring containment facts. A first requirement would be to mathematically characterize the approach:

1. *soundness.* Each rewriting rules in $\mathcal{O}$ must preserve the semantics of paths. This can be proved through a case-by-case analysis of each rule, provided a sound containment relation. The relation $p_1 \succ p_2$ says that p_2 is more optimized (smaller) than p_1. We will come back on this point later.

$$\forall x, p_1, p_2 \;\; p_1 \xrightarrow{\mathcal{O}^+} p_2 \;\; \Rightarrow \;\; \mathcal{S}[\![p_1]\!]_x = \mathcal{S}[\![p_2]\!]_x \wedge p_1 \succ p_2$$

2. *termination.* As a rewriting system, termination is an essential property. However, the current system seems easy to handle, for instance through a syntactic complexity measure, shown to be monotonic decreasing (see [7]). It could become difficult, depending on forthcoming rules we could add into the current system. Note that this issue is tightly related to the last point (see *completeness* below), since the existence of an optimal form requires the finiteness of the computation.

$$\forall p_1, p_3 \;\; \exists p_2 \;\; p_1 \xrightarrow{\mathcal{O}^*} p_2 \Rightarrow \;\; \neg(p_2 \xrightarrow{\mathcal{O}} p_3)$$

3. *confluence.* Such a joinability property is quite useful in order to reason on uniqueness of optimal forms. At least, uniqueness up to an equivalence relation $\approx$ is required, otherwise our system would be fragile, as the result would arbitrarily depend on the choice and the application order of rules.

$$\forall p_1, p_2, p_3 \quad p_1 \xrightarrow{\mathcal{O}^*} p_2 \;\wedge\; p_1 \xrightarrow{\mathcal{O}^*} p_3 \quad \Rightarrow \quad p_2 \approx p_3$$

4. *completeness.* Do we have enough rules in $\mathcal{O}$ in order to handle all cases ? Up to now, we only followed a mathematical intuition. The formal completeness can be defined as

$$\forall x, p_1, p_2 \;\; \mathcal{S}[\![p_1]\!]_x = \mathcal{S}[\![p_2]\!]_x \;\wedge\; p_1 \succ p_2 \;\; \Rightarrow \;\; p_1 \xrightarrow{\mathcal{O}^+} p_2$$

The three first properties are mandatory in order the system to be of practical interest. The forth one is ambitious. If not complete (or not proved so!), the system must however cover a significant amount of cases in order to be realistic. Beyond characterization, the reader may object that any runtime optimization is tightly bound to evaluation algorithms, execution architecture and data structures, and thus can not be stated on the basis of an abstract description. Our goal is to focus on *general* optimization rules for *any* implementation. However, this notion is controversial, as the boundaries might be hard to shape. Let us consider the following optimization step

$$\text{child::}a \;\mid\; \text{child::}*/\text{desc::}a \rightarrow \text{descendant::}a$$

Do we actually always perform an optimization? On one hand, we probably gain the evaluation cost related to the union operator, and also the memory cost of storing intermediate results; On the other hand, the operator might be not directly implemented (e.g. could be a pipe between two concurrent processes), and the memory gain could be even much fuzzy, as we cannot state any algorithmic hypothesis about storage. More interestingly, what is undoubtedly gained in such a rewriting step is redundancy: child::a and child::$*$ both require scanning all children of the context node, and this has an execution and/or a memory cost, whatever used algorithm and architecture. This notion is roughly captured in [11] through the notion of minimal pattern. The authors define pattern minimality as the non-existence of recovering sub-patterns. We propose a simpler notion of minimality, using a trivial function $|p|$ that measures the syntactic complexity of a term p. Such a function from paths or qualifiers into natural numbers can be defined as follows:

Definition 1. *Syntactic complexity measure of paths and qualifiers*

$$\forall p, p_1, p_2, q, q_1, q_2, a, N$$

$$\begin{array}{lcl}
|\bot| = |\wedge| = |\textbf{true}| = |\textbf{false}| & = & 1 \\
|a::N| & = & 2 \\
|p_1/p_2| = |p_1 \circ p_2| = |p_1 \cap p_2| & = & |p_1| + |p_2| + 1 \\
|p[q]| & = & |p| + |q| + 1 \\
|\textbf{not } q| & = & |q| + 1 \\
|q_1 \textbf{ and } q_2| = |q_1 \textbf{ or } q_2| & = & |q_1| + |q_2| + 1 \\
|p_1 \sqsubseteq p_2| & = & |p_1| + |p_2| + 1
\end{array}$$

Now we are able to define the ordering relation required for the characterization of our rewriting system $\mathcal{O}$

Definition 2. *Minimality Order.*

$$\forall p_1, p_2 \qquad p_1 \succ p_2 \quad \textit{iff} \quad |p_1| > |p_2|$$

5. PRELIMINARY EMPIRICAL ANALYSIS

In this section, we describe a preliminary set of experiments. Our goal is to check if a particular containment-based optimization remains useful when evaluated by a real world XPath engine, that probably applies internally various optimization techniques, e.g. caching or case based simplification. The experiments compute the XPath expression of example 1:

$$\wedge//a[*/b/c \text{ and descendant::}b]$$

and its optimized variant on the same set of documents. This one is randomly generated (but complying to precise construction rules) in order to cover a significant variety of trees. The reader may find further details on our experimental protocol in the appendix B.

We defined three different factors: maximum branching width, maximal depth of the tree, and maximal cardinal of the node label set. Each of the three experiments measures the time response depending on the variation of one particular factor. We ran both queries five times on each document sample and the results were averaged (standard deviation

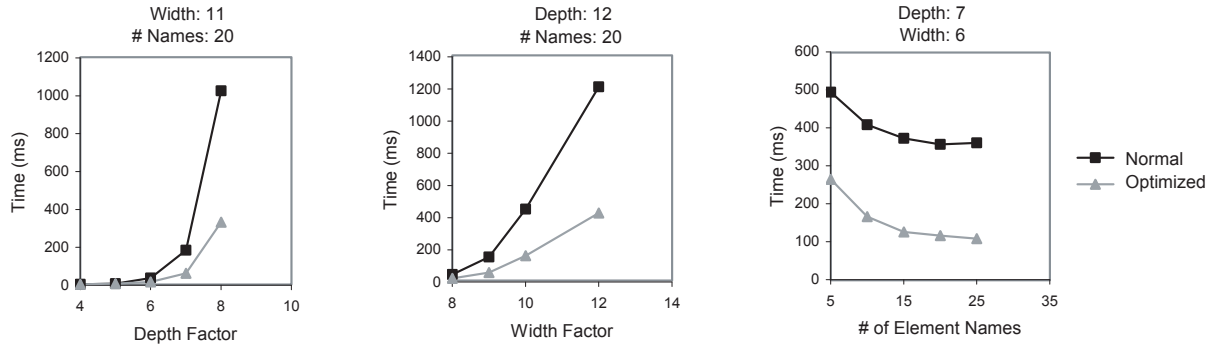

Figure 10: Varying Factors Independently (Xalan C++)

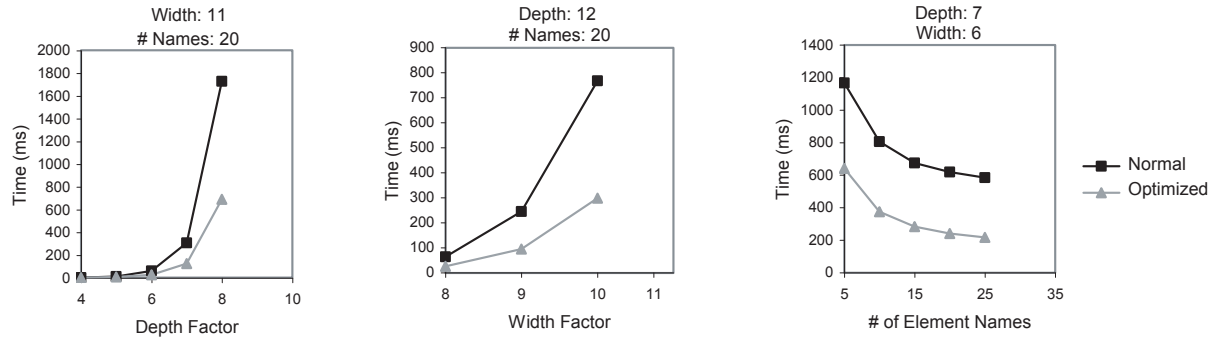

Figure 11: Varying Factors Independently (Libxml)

was observed as a criterion of the experiment's apparatus quality). The final result is built from the average of five runs on five random trees. Figures 10 and 11 summarize the test results for two different XPath engines: "Xalan C++" [19] and "Libxml" [12]. In the first experiment, we varied the depth factor and fixed the maximum possible width factor to be 11^2 and the number of element names to be 20^2. The results are captured on the leftmost graph of each figure. The middle graphs capture the results of the second experiment, when we varied the maximum width factor and fixed the depth factor to be 12 and the number of element names to be 20. Finally, the rightmost graphs show results of the third experiment, when we varied the number of element names and fixed the maximum depth and width factor to be 7 and 6, respectively.

The results show that the optimization is working best for deeper and wider trees. In the first and second experiments, although query performance degrades exponentially (as the size of the tree increases exponentially), the non-optimized query time increases much faster than the optimized query time. For very deep trees, when the depth reaches 8, the optimization provides a three-fold increase in query performance using Xalan C++, as well as for very wide trees, when the width reaches 12. The third experiment confirms that the optimization improves performance when the query retrieves either small or large result sets.

Experiments show that containment-based optimization can improve query performance for large, deep, and bushy trees, wether the query has a high match probability or not. The experiments confirm the interest of the static optimization we propose - at least for the particular case we considered - in the sense that XPath engines do not achieve at runtime any equivalent optimization.

[2]See Appendix B for the definitions of factor units.

6. CONCLUSION AND PERSPECTIVES

In this paper, we have described a static analysis framework for XPath optimization, in which we have proposed to use conditional rewriting rules based on the containment relation for XPath expressions. The optimization aims at eliminating implicit redundancies and contradictions found in user-defined or generated XPath queries. Preliminary experiments show that this framework could provide an interesting basis for a static optimization layer, that could be either implemented on top of an XPath engine, or built inside a compiler.

Further on-going work on the issues explored in this paper is worth mentioning. First, we are working on the completeness issue for the containment inference system. Proving this important property would allow us to state that we can determine whether the containment relation holds or not between any two XPath expressions. Note that even an incomplete containment could be of practical interest as well, especially if the causes are well understood. Second, our optimization technique deserves further investigation for characterizing an optimality notion for XPath expressions (as briefly mentioned in section 4), and ensuring that all kinds of redundancies and contradictions can be detected. In addition, the continuity of this work would require looking at decision issues and algorithmic complexity.

Finally, the real world application of this work relies on our ability to extend the XPath fragment we consider. The next step - our underlying goal - is to address potential scalability issues. The introduction of the **count**(p), **position**() and **last**() primitives in qualifiers are of very first importance, both for the expressive power and for showing the potential of the logic-based framework we are working on. Handling quantitative assertions seems quite tractable using a Presburger-like arithmetic (known to be decidable) together with rules such as:

$$\frac{p_1 \leq p_2}{\textbf{count}\ (p_1) \leq \textbf{count}\ (p_2)} \qquad \frac{}{0 \leq \textbf{count}\ (p)}$$

Another promising direction would be to consider structural constraints on documents (so-called *schemas*) in order to handle more precise containment assertions in the same conceptual framework.

7. REFERENCES

[1] S. Abiteboul and V. Vianu. Regular path queries with constraints. In *Proceedings of the 16th ACM Symposium on Principles of Database Systems*, pages 122–133, May 1997.

[2] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Minimization of tree pattern queries. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 497–508. ACM Press, 2001.

[3] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon. XML path language (XPath) 2.0, W3C working draft, August 2003. http://www.w3.org/TR/2003/WD-xpath20-20030822.

[4] J. Clark. XSL transformations (XSLT) version 1.0, W3C recommendation, November 1999. http://www.w3.org/TR/1999/REC-xslt-19991116.

[5] J. Clark and S. DeRose. XML path language (XPath) version 1.0, W3C recommendation, November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.

[6] The coq proof assistant. http://coq.inria.fr.

[7] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–320. North-Holland, 1990.

[8] A. Deutsch and V. Tannen. Containment of regular path expressions under integrity constraints. In *Knowledge Representation Meets Databases*, 2001.

[9] G. Dowek. Introduction to proof theory. http://www.lix.polytechnique.fr/~dowek/.

[10] D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 formal semantics, W3C working draft, August 2003. http://www.w3.org/TR/2003/WD-xquery-semantics-20030822/.

[11] S. Flesca, F. Furfaro, and E. Masciari. Minimization of tree patterns queries. In *Proceedings of the 29th VLDB Conf.*, pages 497–508, January 2000.

[12] Libxml2. http://www.xmlsoft.org/.

[13] G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 65–76. ACM Press, 2002.

[14] F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *Proceedings of the 9th International Conference on Database Theory*, pages 315–329. Springer-Verlag, 2002.

[15] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *Proc. of the EDBT Workshop on XML Data Management (XMLDM)*, volume 2490 of *LNCS*, pages 109–127. Springer, 2002.

[16] J.-Y. Vion-Dury and N. Layaïda. Containment of XPath expressions: an inference and rewriting based approach. In *Extreme Markup Languages*, Aug. 2003. http://wam.inrialpes.fr/publications/2003/xtrem2003/xtrem2003.pdf

[17] P. Wadler. Two semantics for XPath. http://www.research.avayalabs.com/user/wadler/papers/xpath-semantics/xpath-semantics.pdf, Jan. 2000.

[18] P. T. Wood. Containment for XPath fragments under DTD constraints. In *In Proc. of the 9th International Conference on Database Theory*, pages 300–314, August 2003.

[19] Xalan-c++ version 1.7. http://xml.apache.org/xalan-c/.

APPENDIX

A. XPATH SEMANTICS

Two functions $\mathcal{S}$ and $\mathcal{Q}$ respectively define the semantics of paths and qualifiers, provided a context node x in the tree:

$\mathcal{S} : \mathit{Path} \longrightarrow \mathit{Node} \longrightarrow \mathrm{Set}(\mathit{Node})$

$$
\begin{array}{lcl}
\mathcal{S}[\![\wedge]\!]_x & = & \{x_1 \mid x_1 \rightarrowtriangle^+ x \wedge \mathrm{root}(x_1)\} \\
\mathcal{S}[\![\bot]\!]_x & = & \emptyset \\
\mathcal{S}[\![p_1 \mid p_2]\!]_x & = & \mathcal{S}[\![p_1]\!]_x \cup \mathcal{S}[\![p_2]\!]_x \\
\mathcal{S}[\![p_1 \cap p_2]\!]_x & = & \{x_1 \mid x_1 \in \mathcal{S}[\![p_1]\!]_x \wedge x_1 \in \mathcal{S}[\![p_2]\!]_x\} \\
\mathcal{S}[\![p_1/p_2]\!]_x & = & \{x_2 \mid x_1 \in \mathcal{S}[\![p_1]\!]_x \wedge x_2 \in \mathcal{S}[\![p_2]\!]_{x_1}\} \\
\mathcal{S}[\![(p)]\!]_x & = & \mathcal{S}[\![p]\!]_x \\
\mathcal{S}[\![p[q]]\!]_x & = & \{x_1 \mid x_1 \in \mathcal{S}[\![p]\!]_x \wedge \mathcal{Q}[\![q]\!]_{x_1}\} \\
\mathcal{S}[\![\mathrm{a}::N]\!]_x & = & \{x_1 \mid x_1 \in f_a(x) \wedge \mathcal{T}_a(x_1, N)\}
\end{array}
$$

$\mathcal{Q} : \mathit{Qualifier} \longrightarrow \mathit{Node} \longrightarrow \mathit{Boolean}$

$$
\begin{array}{lcl}
\mathcal{Q}[\![\textbf{true}]\!]_x & = & \mathrm{true} \\
\mathcal{Q}[\![\textbf{false}]\!]_x & = & \mathrm{false} \\
\mathcal{Q}[\![q_1 \textbf{ and } q_2]\!]_x & = & \mathcal{Q}[\![q_1]\!]_x \wedge \mathcal{Q}[\![q_2]\!]_x \\
\mathcal{Q}[\![q_1 \textbf{ or } q_2]\!]_x & = & \mathcal{Q}[\![q_1]\!]_x \vee \mathcal{Q}[\![q_2]\!]_x \\
\mathcal{Q}[\![(q)]\!]_x & = & \mathcal{Q}[\![q]\!]_x \\
\mathcal{Q}[\![\textbf{not } q]\!]_x & = & \neg \mathcal{Q}[\![q]\!]_x \\
\mathcal{Q}[\![p_1 \sqsubseteq p_2]\!]_x & = & \mathcal{S}[\![p_1]\!]_x \subseteq \mathcal{S}[\![p_2]\!]_x
\end{array}
$$

The navigational semantics of axes relies on the relation $\rightarrowtriangle$ that maps a node to its children in the XML tree, and on the transitive closure $\rightarrowtriangle^+$ of this relation; the function $\mathcal{T}$ performs a node test (see the tables below)

a	$f_a(x)$
self	$\{x\}$
child	$\{x_1 \mid x \rightarrowtriangle x_1\}$
descendant	$\{x_1 \mid x \rightarrowtriangle^+ x_1\}$
descendant-or-self	$\{x\} \cup \{x_1 \mid x \rightarrowtriangle^+ x_1\}$
attribute	$\{x_1 \mid x \rightarrowtriangle x_1 \wedge \mathrm{attribute}(x_1)\}$
namespace	$\{x_1 \mid x \rightarrowtriangle x_1 \wedge \mathrm{namespace}(x_1)\}$

N	a	$\mathcal{T}_a(N, x)$
n		$\mathrm{name}(x)=n$
$*$	attribute	$\mathrm{attribute}(x)$
$*$	namespace	$\mathrm{namespace}(x)$
$*$	other	$\mathrm{element}(x)$
text()		$\mathrm{text}(x)$
comment()		$\mathrm{comment}(x)$
processing-instruction()		$\mathrm{pi}(x)$
element()		$\mathrm{element}(x)$
node()		true

B. EXPERIMENTAL ENVIRONMENT

We conducted experiments on a laptop PC (IBM Thinkpad T22). It has an Intel(R) Pentium(R) 3 CPU 900Mhz, 256MB RAM and a 56GB (7200 RPM) hard drive. The PC runs Windows XP Professional version 2002 SP1. We installed the binaries versions of Xalan C++ 1.7.0 and Libxml 2.6.4 for win32.

B.1 Test Programs

Our test program uses static linking to call the original Libxml2 and Xalan DLLs, found in the distributions. The test programs basically initialize the XPath engine, and then determines the time spent to evaluate an XPath expression by recording the system time before and after a call:

- to the method "xmlXPathEvalExpression()", in charge of the evaluation when using Libxml;
- to the method "evaluate ()" of the class "XPathEvaluator" when using Xalan.

The test programs were compiled with Microsoft Visual C++ 6.0 in release mode (debug mode switched off). We isolated the machine for testing. Only the test program and normal operating system background processes are running during the testing period.

B.2 Random Experiment

We generated random XML documents for testing with the following configurable parameters:

- The *depth factor* represents the level of nesting of elements in the XML document, in order to model the recursive structure of a document model. The depth factor controls the depth of subtrees and can be fixed or randomly chosen from a range.
- The *width factor* describes the number of children of a non-leaf node in the tree, in order to model the bushiness of an XML instance. The width factor can be fixed or randomly chosen from a range.
- The *number of element names* describes the number of possible element names, in order to model the cardinality of a namespace allowed by a document model. This factor is fixed.

The tree is made random in two ways:

1. In order to capture a broad range of different topologies, the depth and width of any subtree is made random to test with short, bushy trees, or deep, skinny trees or some combination thereof. Because of limited memory, we considered depth and width factors up to 12. The sizes of the smallest and largest XML documents used for testing were respectively 1KB and 15MB.
2. Each node in the tree can become any element from the allowed range with the same probability, e.g. if the number of element names is 26, a node in the tree will either be named "a" or "b"... or "z" in an equiprobable way. This allows us to vary the query match probability from low to high, in order to test when XPath expressions have a higher chance to match and retrieve large result sets, or when queries have a lower chance to match and retrieve small result sets, or some combination thereof.