Improving Efficiency of XPath-Based XML Querying

Pierre Genevès
INRIA Rhône-Alpes

March 8, 2004

Abstract

XML is becoming the de facto standard for information exchange. XML querying is a key component for structured information processing and plays a central role in the next generation world wide web, information management systems and databases. Applications relying on XML processing notably depend on XPath, the standard language for adressing parts of XML documents. Besides its fundamental functionality, reasons behind XPath success include being widely accepted by programmers and well-suited for formal treatments. With the growing volume of XML content and XML processing applications, our research is oriented toward efficiency of XML querying. Our approach relies on analysis and transformation of XPath expressions for optimization. This note presents current open issues with XPath, and introduces our preliminary results applied to streaming XPath processing. Moreover, it describes our methodology, which includes XPath modelisation using the Coq proof assistant, and future directions envisioned toward high performance XML querying.

1 Introduction

XML is now the dominating standard for representing data and structured documents. Many applications use this representation and subsequently rely on XML processing. Specific solutions for XML processing and notably XML transformation languages have been recently designed, the W3C "XSLT" [13] standard being definitely the most well-known. This experience has shown the crucial role that XML content querying plays in major features of tomorrow's computing infrastructure, like the next generation world wide web, structured documents, information management, and databases.

A response to the need for XML querying has been found with XPath [14]. XPath was introduced as part of the XSLT transformation language to have a non-XML format for selecting nodes and computing values from XML document trees. Since then XPath has become part of several other standards such as XML Schema [16] or XLink [15]. The forthcoming XPath Version 2.0 [11] forms the "navigation subset" of the also forthcoming XQuery XML database access language [12]. Besides its fundamental functionality, reasons behind XPath success include being widely accepted by programmers and well-suited for formal treatments.

Objectives. As more and more XML applications are found, the global amount of XML content is rapidly growing, and XML documents become themselves larger and larger. Efficient XML content querying is crucial for the performance of almost any XML processing architecture. As XPath is being widely adopted for current and forthcoming XML-related standards, the performance of implementations of these languages depends directly on the underlying XPath engine. Thus, improving the performance of evaluating common XPath expressions is essential.

Outline. In Section 2 we introduce XPath for the novice and summarize open theoretical questions related to XPath performance issues. The next Section 3 detail our approach and methodology, before we present preliminary results and discuss applications in Sections 4 and 5, respectively. Section 6 briefly discuss future directions of our work before we conclude in Section 7.

2 XPath and Beyond

2.1 XPath

In their simplest form XPath expressions look like "directory navigation paths". For example, the XPath

$$/\text{company}/\text{staff}/\text{employee} \tag{1}$$

navigates from the root of a document through the top-level "company" element to it's "staff" child elements and on to it's "employee" child elements. The result of the evaluation of the entire expression is the *sequence* of all the "employee" elements that can be reached in this manner, returned in the order they occurred in the document. At each step in the navigation the selected nodes for that step can be filtered with a predicate that tests previous step's selection. So if we ask for

$$/\text{company}/\text{staff}/\text{employee}[2] \tag{2}$$

then the result is *all* employee elements that are the *second* employee element among the employee child elements of each staff element selected by the previous step.

The situation becomes extra interesting when combined with XPath's capability of searching along "axes" other than the shown "children of" axis. Indeed the above XPath is a shorthand for

$$/\text{child::company}/\text{child::staff}/\text{child::employee}[\text{position}() = 2] \tag{3}$$

where it is made explicit that each *path step* is meant to search the "child" axis containing all children of the previous context node, and that a numeric index is really a shorthand for a *predicate* that tests the position number. If we instead asked for

$$/\text{child::company}/\text{child::staff}/\text{child::employee}/\text{following-sibling::}*[\text{position}() \leq 2] \tag{4}$$

then the last step selects nodes of any kind that are in the first two sibling positions immediately *after* each employee.

XPath navigation capabilities include a full variety of axes and notably allow powerful predicate filtering with boolean connectors. For a gentle introduction to XPath see one of the numerous books on XSLT; for a more formal presentation see [10].

2.2 Open Questions

Even though heavily used by its incorporation into a variety of XML-related standards, many questions still remain open around XPath. This section gives an overview of open theoretical questions related to XPath performance issues.

Operational Semantics. A major category of open questions is related to finding adapted semantics for architectures with particular operational requirements. The way XPath is defined in [10], using denotational semantics, directly motivated implementation approaches; thus bypassing the role of more operational semantics, standing between the denotational one and the

implementation. As a consequence, today's major XPath engines are ill-equiped for architectures with specific performance needs. For example, a frequent implicit requirement of today's algorithms is that the whole document is required to be loaded in memory in order to evaluate a query. What happens with large or infinite XML documents (XML flows)? Finding adapted operational semantics for particular architectures with clearly stated hypotheses remains an open issue.

Complexity. Another open question is the precise cost of evaluating an XPath query, which is not yet well understood, even on finite XML documents. Today's major XPath engines that embed an algorithm from the naive class lead to exponential-time processing in the size of the input queries. A polynomial-time algorithm for XPath processing, with respect to combined complexity (i.e. both in terms of the size of the XML data and the input query), has been recently proposed [3]. From the standpoint of theory, the precise complexity of XPath query evaluation is open. Another related question is the parallelization of XPath evaluation. A study of the combined complexity of various XPath fragments [4] has shown that a fragment of XPath 1 can be massively parallelized, but it is still unknown whether the general query evaluation problem can be parallelized.

Equivalence Rules. A possible approach to optimize query evaluation is to find ways for simplifying common XPath expressions patterns. In this direction, some equivalence preserving rewriting rules, relying upon XPath symmetry, have been proposed in [8]. More general is the equivalence problem for XPath expressions: finding if the interpretations of two different XPath expressions always lead to the same set of nodes (e.g. for all document trees and context-state). This problem has been shown to be reducible in polynomial time [6] to another hot question: the containment problem for XPath expressions.

Containment. The containment problem consists in determining wether the interpretation of an XPath expression p_1 (e.g. the node set selected by p_1) is always included in the interpretation of a different XPath p_2. The containment problem quickly becomes undecidable when considering large XPath fragments. A classification of the complexity of the containment problem has been given in [7]. Constructive approaches and practical tools to decide the containment problem over specific XPath fragments would be a salient issue. A promising approach is using rewriting techniques to first attempt to normalize XPath expressions in a form which is more suited for comparison [9].

Document Type Consideration. General XPath equivalences are notably hard to find since they are intended to be valid for all document trees. Considering additional hypotheses, like a document type, allows to restrict the domain of trees. A few methods have been proposed for rewriting XPath expressions by considering the type of the document on which the query operates [5]. However, such work relies only on partial document type schemas, far from the expressive power allowed by XML Schemas based descriptions.

3 Approach: XPath Analysis and Transformation

3.1 Static Improvement of Performance

Our approach is intended to be applicable with any XPath-based XML standard; it is based on static analysis and transformation of XPath expressions occuring in these standards. We are concerned with accuracy assessment and optimization on the basis of the characteristics

of the static model design and source code, prior to machine execution. For example, if we consider an XSLT stylesheet, this could mean extracting all XPath expressions from it, then statically rewrite them in order to output a different XSLT stylesheet. The new stylesheet is such that additional properties are verified on it. Our approach is generic in the sense it allows to optimize a stylesheet for different particular purposes. For example, the interpretation of the final stylesheet is more efficient, its compilation outputs a module whose performance is adapted for specific architecture requirements; or the queries in the stylesheet comply with a given XML Schema.

3.2 A Generic Platform for Static Analysis

Using langage theory, rewriting and logics, we attempt to develop adapted formal methods to work with the XPath standard. The first faced issue is the combinatorial complexity of XPath. Therefore, before studying particular problems, our work currently focus on obtaining a normal form for XPath expressions, more suited for analysis operations. For example, the normal form is disjunctive, stratified, and verifies additional strong properties like having one and only one qualifier for each step. All these properties can be used as hypotheses in order to reduce combinatorial aspects encountered for analysis operations. We believe our work will provide a foundation for any problem that implies analysis operations on XPath, such as the containment problem, and specific optimizations of XPath expressions for particular purposes.

3.3 A Practical, XPath-Centric Vision

Our approach can be opposed to other model-based approaches that attempt to transpose XPath into some low level model (usually relying on automata). The drawbacks of these approaches are the difficulties encountered to transpose the result back to the XML or XPath world. These approaches would hardly allow to output modified XPath expressions as illustrated in the stylesheet transformation scenario above. As opposed to these model-based approaches, our approach aims at placing XPath semantics at the center of concerns.

3.4 Methodology

Incremental Analysis. Since XPath analysis is heavy combinatorial, model-based approaches and other related work (summarized in Section 2.2) usually consider small subsets of XPath 1; very restricted in reguard of the forthcoming XPath 2.0 standard. Subsequently, there is often a big gap between the applicative scope of theoretical results and the actual XPath expressions used by programmers. As opposed to reducing XPath in order to transpose it into a well-known model, we try to adopt a more specific and constructive approach, by considering an XPath fragment that we extend progressively. Besides enriching the fragment toward a practical use of the full standard, this incremental analysis also helps at identifying precisely where complexity issues come from.

Formal Correctness. Most static transformations must preserve the initial meaning of the expression. In order to check that a transformed XPath expression is correct, given a formal semantics $\mathcal{S}$ of XPath [10], we have to prove the formal correctness of our transformations. Given an XPath p and a context node x, if $\mathcal{S}[\![p]\!]\,x$ specifies the set of nodes selected by p, our transformation T is correct if:

$$\forall x : \mathcal{S}[\![p]\!]\,x = \mathcal{S}[\![T(p)]\!]\,x \tag{5}$$

where "=" is the set equality.

The proof is usually done by structural induction as $\mathcal{S}$ is inductively defined. However, such a proof can become difficult to maintain for combinatorial reasons as our XPath fragment gets more and more syntactic constructs, and as the transformation T becomes more complex, too.

Use of a Logical Framework. To construct and help us manage such proofs we chose to use Coq [17]. Coq is a proof assistant for a logical framework known as the calculus of inductive constructions. Coq allows the interactive construction of formal proofs, and also the manipulation of functional programs consistently with their specifications. Once we described the formal semantics of XPath, Coq has shown to be useful to prove the soundness of our transformations. We notably define specific tactics and use tactics composition to reduce significantly the combinatorial aspects. Moreover, by allowing to save and "replay" proofs, Coq also helps to maintain the global consistency by updating the proof term as we slightly change the goal or hypotheses.

4 Preliminary Results: Context State Elimination from XPath

A peculiarity of the XPath semantics. The result of an XPath 2.0 expression is a *sequence* instead of a set as in XPath 1.0. A sequence is an ordered collection of zero or more items that can be XML document nodes or atomic values. A peculiarity of the XPath semantics is that every expression is defined in terms of a dynamic context, mainly based on:

- The *context item* which is the item currently being processed in a path step. When an expression $p[q]$ is evaluated, each node resulting from the evaluation of p becomes the *context item* for the evaluation of q.
- The *context position* is the position of the *context item* within the sequence of items currently being processed.
- The *context size* is the cardinal of the node sequence currently being processed.

XPath queries often contain "position()" and "last()" that respectively return the *context position* and *-size*. As they refer to the dynamic context they are called *context-sensitive expressions*.

A limitation for XPath evaluators. In standard interpretations models directly inspired from XPath semantics in [10], an XPath query is evaluated by a top-down traversal of the XPath expression tree. The input XML tree is traversed according to each sub-expression and the context state is updated along the way. We identified two drawbacks in this approach:

1. the run-time system needs to maintain the context state at all times, in case it is accessed. This approach leads to significant overhead since the context may be irrelevant to subsequent evaluation steps ;
2. having to keep track of the context state is a general limitation for implementations. For example, context-size references block streaming XPath evaluation, as we will see in further details in section 5.1.

XPath Context State Elimination. Our preliminary results consist in showing how to bypass these limitations. We showed how to statically rewrite an XPath query that involves

context-sensitive expressions into an equivalent query that does not [2]. Let us consider the example 3: the position among the employee children of each staff parent (node) can be computed relatively to the child by counting the preceding siblings. Specifically (3) is equivalent to

$$/\text{company}/\text{staff}/\text{employee}[\text{count}(\text{preceding-sibling::employee}) + 1 = 2] \tag{6}$$

More precisely, it turns out that the position can be calculated from expressions relatively to the current node and the node that generated the innermost sequence. This is because each of these two nodes defines a clean partitioning of the complete collection of nodes. To refer multiple times to a sub-expression inside the same XPath query, we extended the expressive power of XPath with a "let... return..." construction borrowed from XQuery [12]. For more formal details, as well as the complete algorithm for translating context references that can occur in XPath 2.0 expressions, the reader can refer to [2] .

The main application of this is to allow a new class of stateless implementations. These implementations do not need to keep track of context information that might be never used. Moreover, this new class becomes extra interesting for streaming XPath processing, as we will see in the next section.

5 Applications: Improving XPath Performance

5.1 Allowing Particular Architectures

XML flows querying. In applications such as financial data analysis, digital TV broadcasting, telecommunications data management, network monitoring, manufacturing, sensor networks, and others, data takes the form of continuous XML streams rather than finite stored XML documents, and clients require long-running continuous queries as opposed to one-time queries. In these particular data-driven architectures, the application cannot seek forward or backward in the XML tree, nor can it revisit a node encountered earlier unless it is explicitly buffered. Traditional evaluation techniques require the entire document to be in memory which is not adapted for such particular architectures.

Streaming is also interesting for processing huge documents, instead of traditionnal evaluation techniques that may result in unacceptable overhead. Processing large XML documents on the fly is useful because of the greater efficiency of streaming systems (which use a sequential scan instead of nonsequential data access on disk).

XPath and streaming XML. Main problems that prevent from using directly XPath queries on streaming XML come from XPath powerful navigationnal capabilities. More precisely reverse axes allow to search backward in the tree, and context-sensitive expressions that potentially refer to an incomplete context. A few techniques have been proposed for removing reverse axes by rewriting them into forward ones [8, 1]. A remaining issue is caused by XPath context-sensitive expressions. Consider for example an XPath expression of the form:

$$p_1[\text{last}() \geq 3]/p_2 \tag{7}$$

This expression cannot be progressively evaluated against an XML stream since it contains "last()", a context-sensitive expression. Indeed, in this expression, "last()" refers to the cardinal of the sequence selected by p_1. The complete sequence is theoretically unknown until the "end of the stream", thus "last()" would block the evaluation. Using our preliminary results, "last()" can be eliminated from (7): (7) can be rewritten into a new expression (that depends on the form of p_1) and that does not involve any context reference. Thus, it becomes possible to output results of (7) on the fly.

Our preliminary results, coupled with techniques that rewrite reverse axes, enable and optimize the use of context-sensitive expressions in a stream-based context, e.g. for data-driven architectures. Applications of future results include allowing or enhancing the use of XML querying for other kind of particular architectures, such as distributed environments.

5.2 General Performance

Apart from particular architectures with specific execution requirements, applications also aim at improving the efficiency of the general XPath performance. Indeed, short-term expected results of our approach include providing a common basis for open theoretical questions summarized in Section 2.2. Using our logical framework, any problem that implies analysis operations on XPath, such as the containment problem can be worked out with a new constructive approach. Applications of the containment problem include optimizations of XPath expressions. Consider for example an XPath expression of the form $p_1|p_2$, if $p_1 \leq p_2$ then there is no need to evaluate p_1, and $p_1|p_2$ can be optimized by being rewritten into p_2.

From a longer-term point of view, we believe our approach could be a first step toward a software tool that would check and optimize XPath queries; a tool which is still missing today for popular XML standards such as XSLT.

6 Future Directions

As an experimental direction for future XPath optimizations, we plan to take the content model into account for rewriting XPath queries. The idea is to specialize XPath expressions according to a known document type, as briefly mentioned in section (2.2). Consider for example the XPath expression

$$/\text{company}/(\text{staff}|\text{site}) \tag{8}$$

and a document type that restricts the child elements of "company" elements to be either "staff" or "site" elements. In this case, the query (8) can be rewritten into /company/$*$, equivalent from a denotational point of view. This simple example raises the more general question of knowing when and how it is pertinent to use document types for XPath optimization.

7 Conclusion

In this note we have presented our research oriented toward efficient XML querying performance. We detailed our approach which involves the construction of a generic logical framework adapted for formal work on XPath performance issues. After discussing how our XPath-centric approach differs from others, we explained our methodology. We then summarized our preliminary results for eliminating context state from the W3C XPath standard, and explained how they can be applied for streaming XPath processing. This particular application for data-driven architectures lies within the scope of global applications of our research, which aims at improving XPath-based XML querying performance.

References

[1] C. Barton, P. Charles, D. Goyal, M. Raghavachari, V. Josifovski, and Marcus F. Fontoura, *Streaming XPath Processing with Forward and Backward Axes*, ICDE - International Conference on Data Engineering, Bangalore, India, March, 2003.

[2] P. Geneves, K. Rose, *Eliminating Context State From XPath*, October 2003, http://wam.inrialpes.fr/people/geneves/xpath-cs-removal.pdf.

[3] G. Gottlob, C. Koch, and R. Pichler, *Efficient Algorithms for Processing XPath Queries*, In Proc. VLDB'02, 2002.

[4] G. Gottlob, C. Koch, and R. Pichler, *The Complexity of XPath Query Evaluation*, In Proc. PODS 2003.

[5] A. Kwong, M. Gertz, *Schema-Based Optimization of XPath Expressions*, http://sirius.cs.ucdavis.edu/publications/KG02a.pdf.

[6] G. Miklau and D. Suciu, *Containment and Equivalence for an XPath Fragment*, Symposium on Principles of Databases Systems, 2002.

[7] F. Neven and T. Schwentick, *XPath Containment in the presence of disjunction, DTDs, and variables*, International Conference on Database Theory, 2003.

[8] D. Olteanu, H. Meuss, T. Furche, F. Bry, *XPath: Looking Forward*, In Proc. of the EDBT Workshop on XML Data Management (XMLDM), 2002.

[9] J-Y. Vion-Dury, N. Layaïda, *Containment of XPath expressions: an Inference and Rewriting based approach*, Extreme Markup Languages, August 4-8, 2003.

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

Recommendations & Working Drafts

[11] A. Berglund, S. Boag, D. Chamberlin, M. 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.

[12] D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, 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/.

[13] J. Clark, *XSL Transformations (XSLT) Version 1.0*, W3C Recommendation, November 1999, http://www.w3.org/TR/1999/REC-xslt-19991116.

[14] J. Clark, S. DeRose, *XML Path Language (XPath) Version 1.0*, W3C Recommendation, November 1999, http://www.w3.org/TR/1999/REC-xpath-19991116.

[15] S. DeRose, E. Maler, D. Orchard, XML Linking Language (XLink) Version 1.0, W3C Recommendation, 2001, http://www.w3.org/TR/xlink/.

[16] H. S. Thompson, D. Beech, M. Maloney, N. Mendelsohn, P. V. Biron, A. Malhotra, D. C. Fallside, XML Schema, W3C Recommendation, 2001, http://www.w3.org/XML/Schema.

Software

[17] The Coq proof assistant, http://coq.inria.fr.