Optimized Translation of XPath into Algebraic Expressions Parameterized by Programs Containing Navigational Primitives

Sven Helmer
helmer@informatik.uni-mannheim.de

Carl-Christian Kanne
cc@informatik.uni-mannheim.de

Guido Moerkotte
moerkotte@informatik.uni-mannheim.de
University of Mannheim
Germany

Abstract

We propose a new approach for the efficient evaluation of XPath expressions. This is important, since XPath is not only used as a simple, stand-alone query language, but is also an essential ingredient of XQuery and XSLT.

The main idea of our approach is to translate XPath into algebraic expressions parameterized with programs. These programs are mainly built from navigational primitives like accessing the first child or the next sibling. The goals of the approach are 1) to enable pipelined evaluation, 2) to avoid producing duplicate (intermediate) result nodes, 3) to visit as few document nodes as possible, and 4) to avoid visiting nodes more than once. This improves the existing approaches, because our method is highly efficient.

1 Introduction

XPath is an essential ingredient of mainstream XML applications like XSLT and XQuery. Moreover, XPath is often used as a simple query language itself. This motivated us to take a close look at efficient evaluation strategies for XPath and come up with a new approach.

Standard XPath evaluators like Xalan or XT, for example, perform an evaluation that processes the location steps in an XPath expression from left to right, step by step. For n nodes this leads to n intermediate results, which are then unioned. During unioning duplicate elimination is performed, the result of which is fed into the next location step. Thereby, the result of every location step is materialized.

Besides other advantages, our approach allows a pipelined evaluation of XPath expressions. Pipelining, which avoids building intermediate results, is one of the major reasons why query evaluation in relational database systems is highly efficient. Our goal was to realize pipelined evaluation of XPath expressions. As we will see, several obstacles need to be overcome. The biggest one is that the results of a location step applied to two different nodes are often not disjoint. Performing a duplicate elimination operation after every location step, however, jeopardizes any attempt at pipelined processing.

1.1 Problem

Let us illustrate the problem by a simple example. As every location step results in a set of nodes, a simple `UnnestMap` operation is the appropriate algebraic operator for representing a location step [8]. We use subscripts to denote the location step the operation refers to. Let l be a location step including a node test and possible predicates. Then $\texttt{UnnestMap}_{\$1:l}(\cdot)$ produces for every input node a set of output nodes that are reachable by l. The result nodes are successively bound to a variable or attribute named \$1. We assume that attributes of tuples handed from one algebraic operator to another may not only have a basic type (like integer or string), but may also be of type node. More formally we can define

$$\texttt{UnnestMap}_{\$i:l}(e) := \{x \circ [\$i : y] | x \in e, y \in x/l\}$$

where \$i is an attribute name, e a general XPath expression, and $\circ$ denotes tuple concatenation. It is important to note that this operator does not eliminate duplicates: as usual in standard relational algebra implementations we assume a bag semantics for our algebraic operators.

We now take a look at a straightforward translation of the XPath expression `//A//A` into a sequence of `UnnestMap` operations:

$$\texttt{UnnestMap}_{\$2:\texttt{desc}_{::A}}(\texttt{UnnestMap}_{\$1:\texttt{desc}_{::A}}(\cdot)).$$

If this expression is applied to the document root node of an XML document that is a binary tree of elements of type

A only, this expression clearly produces duplicates. Moreover, its runtime is quadratic in the number of nodes of the input document. In general, a straightforward translation of an XPath expression results in a sequence of UnnestMap operations, where plenty of duplicates can occur. Since the XPath specification demands set semantics, we need a final duplicate elimination in order to ensure correctness. Still, a lot of duplicate work may be performed by UnnestMap operations following an intermediate result that contains duplicates. Introducing a duplicate elimination operation after every UnnestMap operation avoids the unnecessary work but jeopardizes pipelining, since duplicate elimination is a pipeline breaker.

1.2 Context

The above description of UnnestMap is given on a logical level. In a real implementation, algebraic operators are typically implemented as iterators providing open, next, and close methods [10]. The context of this work is the native XML database system Natix [13]. In Natix, algebraic operators are parameterized by programs written in an assembler-like language which are then interpreted by the Natix Virtual Machine (NVM). XML-related primitives allow simple navigations like getFirstChild, getNextSibling and the like. This is similar to DOM, for example, which also provides navigational primitives. Our approach carries over to all systems that support primitive navigations between nodes.

In Natix, an UnnestMap operator is parameterized by three programs. The first program (called init program) produces the first result tuple. One can think of this program as being called by the open method. The second program (called step program) produces the next tuple of the output. It is called by the next method. Another program (responsible for cleaning up), which is of no importance for understanding this paper, is called by the close method.

1.3 Contributions and Approach

Our contribution is a translation mechanism from XPath expressions to programs for a sequence of UnnestMap operations such that

1. pipelined evaluation becomes possible,
2. no duplicates are produced,
3. as few document nodes as possible are visited,
4. the number of visits per document node is minimized, and
5. the result nodes are returned in document or reverse document order.

In fact, we not only enable pipelined evaluation of XPath expressions, but are also often able to reduce the size of an intermediate result to one node. Although, in principle we are able to treat all axes of XPath, we have to apply some restrictions on the XPath expressions we treat in this paper. The reasons are complexity and space limitations. Nevertheless, the subset of XPath we treat is much larger than the subset containing only child and descendent axes typically treated in literature [2].

We achieve these ambitious goals in three steps. First we eliminate those axes by XPath rewrite that will hamper a smooth translation process. Then we rewrite XPath expressions again by introducing artificial *step functions* to enable further rewrites allowing uncomplicated code generation. In a last step, we translate the rewritten, enhanced XPath expressions into efficient programs.

1.4 Restrictions

The first restriction is that we do not allow position() and last() function calls in our XPath expressions. We do not elaborate on the namespace axis either since it can be treated similar to the attribute axis. The other restrictions can be roughly described as follows. We do not treat the preceding-sibling and following-sibling axes. Although our approach carries over to XPath expressions embedded in predicates, we will not detail their treatment. Last, the prefix of an XPath expression that precedes a following or preceding axis must – in the basic approach – adhere to some restrictions concerning the order in which axes may occur (see Sec. 3 for details). Section 6 will indicate how these restrictions can be lifted. However, we currently do not have an alternative solution to handle the ancestor axis if it is not followed by a following or preceding axis. In this case, we must rely on its straightforward computation followed by duplicate elimination. Note that this fallback to a straightforward evaluation with interspersed duplicate eliminations still remains an option for any case we cannot handle more efficiently.

1.5 Related Work

We are not alone in searching for efficient evaluation techniques of XPath. Several authors have been concerned with rewriting XPath expressions using only the child and descendent axis. They minimize these tree pattern queries [3, 16]. Work along this line can be used to preprocess XPath expressions and it is beneficial to apply these minimizations prior to our translation process. Since backward axes are impossible to evaluate on streaming XML, approaches exist to rewrite XPath expressions such that only forward axes are used [4, 15]. Again, this work is orthogonal to ours. In fact, we will use some of their rewrite rules to preprocess XPath expressions in our first step.

A standard evaluation procedure for XPath expressions that contain only `child` and `descendent` axes is to translate them into an automaton. This approach is described in [2, 5], for example. The focus here is on selective information dissemination. A generalization to more axes is still missing.

Some optimizing rewrites can be applied to XPath expressions when the DTD of the queried documents is known [14]. Again, this work is orthogonal to ours and can be beneficially applied prior to our translation process.

Gottlob et al. propose an evaluation strategy for general XPath expressions in [9]. The underlying idea is the same as in our approach: avoid duplicate evaluation. Their approach is similar to our NVM DupElim approach which pushes duplicate eliminations down the evaluation tree. However, the highest performance gains and the enabling of pipelining is achieved by our rewrite phase. [9] lack such a rewrite phase. Another advantage of our approach is that we produce results in document or reverse document order.

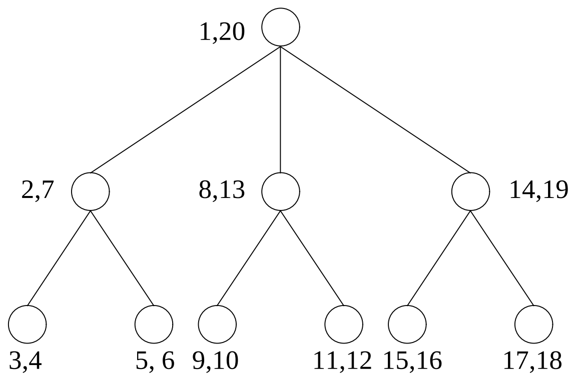
Other evaluation strategies for XPath expressions build on relational representations of XML in conjunction with numbering schemes. One specific approach is to translate XPath into SQL (the latest here is [17]). Others are concerned with providing efficient join algorithms to evaluate a single location step [1].

The functions *first* and *last*, which we will introduce later, resemble some algebraic operators found in [6]. However, they do not consider translation of path expressions into their algebra but require the user to use their algebra as a query language. Also, they do not allow nested objects in their GC-lists, which are used for evaluating the algebra. Context sets in XPath may contain nested objects (e.g. nodes that are descendants of others).

An orthogonal area of research considers the acceleration of XPath evaluation by using indexes [7, 11].

1.6 Organization

The remainder of the paper is organized as follows. In Section 2 we introduce some preliminaries and the first simple rewrite steps to prepare XPath expressions for our purposes. The goal of the preliminary rewrite is to eliminate the axes that are not relevant to our work since their computation does not pose any problems, or the axes that disturb further rewrites. Section 3 introduces the notion of *step functions* that generalize location steps. Using the introduced step functions, we rewrite XPath expressions until they have a convenient form for code generation. The main goal here is to break down step functions in such a way that they operate on single location steps. Section 4 describes how code generation proceeds for the different location path functions. Section 5 presents some performance measurements. Section 6 shows how the restrictions mentioned above can be relaxed such that we are able to translate many more XPath expressions into optimized plans. Section 7 concludes the paper.



Each node is assigned the pair `dmin`, `dmax`.

Figure 1. Depth-first numbering of document nodes

2. Preliminaries and Preparatory Rewrites

2.1. Notation, Abbreviations, Assumptions

We use α and β to denote XPath expressions. Node tests are denoted by n or m. For predicates we use p and q. In order to keep the XPath expressions in the paper short, we use `desc`, `anc`, `prs`, `fos`, `fol`, `pre`, and `par` as abbreviations for `descendant`, `ancestor`, `preceding-sibling`, `following-sibling`, `following`, `preceding`, and `parent`. We abbreviate `-or-self` by `-os`. Whenever we are not interested in the specific node test or predicates, we just omit them from the path expression.

For the introduction of step functions we need a node numbering according to a depth-first traversal. We will denote by `dmin` the number that is assigned to a document node at the time of its first visit during a depth-first traversal. By `dmax` we denote the number that is assigned to a node upon the second (and last) visit. For a simple example document these numbers are given in Fig. 1.

The traversal primitives implemented in the runtime system are assumed to support preorder traversal and its reverse (by an iterator concept). A runtime system supporting postorder traversal will speed up certain evaluation steps. Otherwise we have to emulate postorder traversal with preorder traversal.

2.2. Preparatory Rewrites

Since XPath contains 13 axes, we would like to cut down the number of axes considered in this paper. We do so without loss of generality. There are two problems. The first problem is that some axes produce duplicate nodes, even if the input set does not contain duplicates. This problem originates from the definition of XPath. The second problem has its origins in our approach. For some axes it is not easy to find efficient rewrites. If an axis does not pose any of these two problems, we can safely omit it from further considerations. The axes we can safely omit are `self`, `namespace`, and `attribute`. The `self` axis can be evaluated by applying a simple selection operation. The `namespace` and `attribute` axes do not generate duplicates if their input does not contain duplicates. Furthermore, one can show that it is even possible to rewrite path expressions such that without loss of generality these axes occur only at the end of XPath expressions [12]. In this case, adding an `UnnestMap` operator accessing attributes or namespaces to the plan is straightforward.

The next two axes that are somewhat troublesome are `following-sibling` and `preceding-sibling`. These axes can produce duplicates even if their input does not contain duplicates. Although we are able to treat them (see Sec. 6 and [12]), their evaluation is slightly less efficient than that of other axes. Fortunately, [15] provides rewrites to eliminate `following-sibling` and `preceding-sibling` if they are preceded by a backward axis. We add the following rewrite rules to eliminate further top-level occurrences:

$$\begin{aligned} \texttt{desc} :: n/\texttt{fos} :: m &\equiv \texttt{desc} :: m[\texttt{prs} :: n] \\ \texttt{child} :: n/\texttt{fos} :: m &\equiv \texttt{child} :: m[\texttt{prs} :: n] \end{aligned}$$

Remaining occurrences are handled as indicated in Sec. 6.

Another axis that produces duplicates even if the input set does not contain them is `parent`. Again, occurrences of the `parent` axis within path expressions are not always handled smoothly by our approach. Hence, we eliminate as many occurrences as possible by rewrites. We add to the `parent` elimination rules of [15]:

$$\begin{aligned} &\texttt{anc} :: n[p]/\texttt{par} :: m[q] \\ \equiv\ &\texttt{par}/\texttt{anc} :: m[q \wedge \texttt{child} :: n[p]] \\ &\texttt{anc-os} :: n[p]/\texttt{par} :: m[q] \\ \equiv\ &\texttt{anc} :: m[q \wedge \texttt{child} :: n[p]] \end{aligned}$$

These rules either eliminate occurrences from the top level or move them one step further to the front of the XPath expression where they are possibly eliminated. Occurrences at the very beginning of an XPath expression can be handled easily (as root nodes do not have parents).

There are several optimizing rewrites possible for XPath expressions. We can apply them beneficially before proceeding with our optimized translation procedure. Minimization of occurring tree patterns is one of them [3, 16]. Another example is to replace `desc-os` $::$ $*$/`child` $::$ $x[p]$ by `desc` $::$ $x[p]$. This is an important rewrite, since the first form (abbreviated as $//\texttt{x}[p]$ in XPath) occurs quite frequently. Other techniques such as exploiting DTDs for rewrites (see [14]) should also be applied.

3. Step Function based Rewrite

In terms of the number of nodes visited, the axes `desc`, `desc-os`, `fol`, and `pre` are the most expensive ones. Further, for all input sets with a cardinality larger than one all these axes produce duplicates with a high probability. Hence, we subsequently concentrate on these axes and try to make their evaluation as efficient as possible.

3.1. Step Functions

By definition of XPath, the `following` axis of a node n contains all nodes whose `dmin` value is larger than the `dmax` value of n. Hence, it suffices to compute the `following` nodes of the node whose `dmax` value is minimal. A similar argument holds for computing all the `preceding` nodes of a given set of nodes: it suffices to compute all `preceding` nodes of the node with the maximal `dmin` value.

Let us formalize this idea. For a given set of nodes X and $d \in \{\texttt{dmin}, \texttt{dmax}\}$ we define

$$\begin{aligned} \text{first}_d(X) &:= \{x | x \in X, x.d = min(\{y.d | y \in X\})\} \\ \text{last}_d(X) &:= \{x | x \in X, x.d = max(\{y.d | y \in X\})\} \end{aligned}$$

We call these functions *step functions* since we will treat them like “regular” steps within path expressions.

The semantics of a step function occurring in an (extended) XPath expression is that for every context node the argument of the step function (typically a path expression) is evaluated to a set of nodes. The result of the location step is the result of applying the step function to the set of nodes derived by evaluating the argument. More formally, we define the semantics of step function f occurring in a path expression $\alpha/f(\beta)$ as follows. Let α evaluate to the set X. Then

$$\alpha/f(\beta) := \bigcup_{x \in X} f(x/\beta)$$

where x/β is the set of nodes reachable by the path expression β with x as the current node.

For an arbitrary path expression α, we can make the following two important observations:

$$\alpha/\texttt{fol} \quad = \quad \text{first}_{dmax}(\alpha)/\texttt{fol}$$

$$\alpha/\texttt{pre} \quad = \quad \text{last}_{dmin}(\alpha)/\texttt{pre}$$

Note that if we can compute first_{dmax} efficiently, no duplicates will be generated by computing the subsequent `following` axis. Moreover, instead of computing all result nodes for α, it suffices to compute only one! Both savings can dramatically reduce the evaluation costs of an XPath expression containing a `following` or `preceding` axis.

Let us now turn to the `descendant` and `descendant-os` axis. Their treatment in our approach is identical. The question we have to answer first is in which situations `descendant` or `descendant-os` generate duplicates. This is the case whenever two nodes exist in the input set such that one is the ancestor of the other. If there are no such nodes, we can be sure that we do not produce duplicates. Exploiting this feature leads to the next step function called roots. For a given set of nodes X, it is defined as follows:

$$\text{roots}(X) \quad := \quad \{x | x \in X, \nexists y \in X \;\; y \text{ is an ancestor of } x\}$$

3.2. Rewrite

We have replaced XPath expressions with expressions containing our step functions, which avoids the generation of duplicates. We now rewrite these expressions such that code generation becomes feasible.

For this the following properties of step functions are very useful. Let s be a singleton location step, i.e. a location step that produces for a single input node at most one output node. Then, the following holds:

$$\text{first}_d(s/\alpha) \quad = \quad s/\text{first}_d(\alpha) \qquad (1)$$
$$\text{last}_d(s/\alpha) \quad = \quad s/\text{last}_d(\alpha) \qquad (2)$$
$$\text{roots}(s/\alpha) \quad = \quad s/\text{roots}(\alpha) \qquad (3)$$

Note that s can also be a step function that returns a single node such as first_{dmax}. Another possibility is that s consists of an axis that produces a single node. Typical candidates are `parent` and `self`. Yet another possibility is to use DTD knowledge to infer that an axis can only return a single node. The usefulness of these equations is that we are able to move some steps out of our step functions.

The rules above are a first step, but we need more complex rules to fully rewrite path expressions. Therefore we introduce step functions where possibly large chunks of the original path expression become arguments. Since these are difficult to handle during code generation, more rules are needed to distribute step functions over complex path expressions such that their arguments become less complex path expressions. In case of first and last, we rewrite until their arguments are single steps. For roots we rewrite until the argument path starts with a `descendant-os` or `descendant` axis followed by an arbitrary number of occurrences of `child`.

Let us define the set of down-axes as $D := \{\texttt{desc}, \texttt{desc-os}, \texttt{self}, \texttt{child}\}$ and the set of specially processed axes as $S := \{\texttt{fol}, \texttt{pre}, \texttt{desc}, \texttt{desc-os}\}$. Consider a path expression of the form $\alpha/s/\beta$ where s is the last location step that is a member of S. Then we introduce our step functions by applying one of the following rewrite rules:

$$\alpha/\texttt{fol}/\beta \quad = \quad \text{first}_{dmax}(\alpha)/\texttt{fol}/\beta \qquad (4)$$
$$\alpha/\texttt{pre}/\beta \quad = \quad \text{last}_{dmin}(\alpha)/\texttt{pre}/\beta \qquad (5)$$
$$\alpha/\texttt{desc}/\beta \quad = \quad \text{roots}(\alpha)/\texttt{desc}/\beta \qquad (6)$$
$$\alpha/\texttt{desc-os}/\beta \quad = \quad \text{roots}(\alpha)/\texttt{desc-os}/\beta \qquad (7)$$

If β contains an axis not in D, special care is taken to eliminate duplicates as early as possible (e.g. for the `anc` or `par` axis).

After introducing a step function, its argument is the possibly complex path expression α. Subsequently, we have to simplify α such that code generation becomes possible. We first rewrite the argument of a $\text{roots}(\alpha)$ expression. In doing so, occurrences of first and last may be introduced. We apply the following rules:

$$\begin{aligned} \text{roots}(\alpha/\texttt{fol}/\beta) \quad &= \quad \text{first}_{dmax}(\alpha)/\texttt{anc-os}/\texttt{fos}/ \\ &\quad\quad \text{roots}(\texttt{desc-os}/\beta) \qquad (8) \\ &\quad\quad \text{where } \beta \subseteq D \\ \text{roots}(\alpha/\texttt{pre}/\beta) \quad &= \quad \text{last}_{dmin}(\alpha)/\texttt{anc-os}/\texttt{prs}/ \\ &\quad\quad \text{roots}(\texttt{desc-os}/\beta) \qquad (9) \\ &\quad\quad \text{where } \beta \subseteq D \\ \text{roots}(\alpha/\texttt{desc}/\beta) \quad &= \quad \text{roots}(\alpha)/\text{roots}(\texttt{desc}/\beta) \quad (10) \\ &\quad\quad \text{if } \beta \text{ contains only child axes} \\ &\quad\quad \beta \text{ may be empty} \\ \text{roots}(\texttt{child}/\alpha) \quad &= \quad \texttt{child}/\text{roots}(\alpha) \text{ if } \alpha \subseteq D \, (11) \\ \text{roots}(\texttt{parent}/\alpha) \quad &= \quad \texttt{parent}/\text{roots}(\alpha) \qquad (12) \end{aligned}$$

Eqn. 10 is also valid if `desc-os` is used instead of `desc`. After rewrite, we require that only paths of the form $\texttt{desc}/\texttt{child}^*$ or $\texttt{desc-os}/\texttt{child}*$ occur as arguments of roots. Note that this is not always the case, for example, if `fos` occurs in α and is not followed by `fol` or `pre`.

We now turn to simplifying the argument path of $\text{first}_{\texttt{dmax}}(\alpha)$ and $\text{last}_{\texttt{dmin}}(\alpha)$. The goal is to distribute first_d and last_d occurrences over α such that each of their argument paths has a length of exactly one (location step). In doing so it does not suffice to consider $\text{first}_{\texttt{dmax}}$ and $\text{last}_{\texttt{dmin}}$. We also exploit $\text{first}_{\texttt{dmin}}$ and $\text{last}_{\texttt{dmax}}$. The rewrite rules are

$$\text{first}_{dmax}(\alpha/\texttt{pre}) = \text{first}_{dmax}(\alpha[\texttt{pre}])/\text{first}_{dmax}(\texttt{pre}) \quad (13)$$

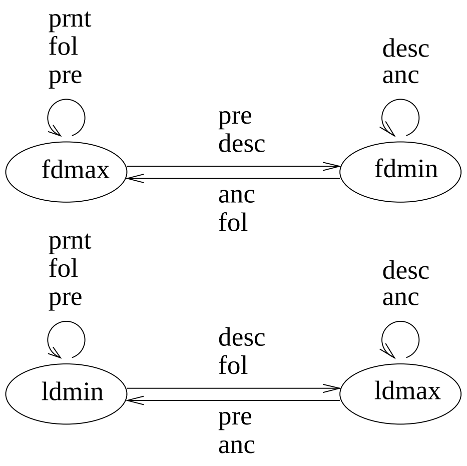


Figure 2. Automaton for first and last

$$\text{first}_{dmax}(\alpha/\texttt{pre}) = \text{first}_{dmin}(\alpha[\texttt{pre}])/\text{first}_{dmax}(\texttt{pre}) \quad (14)$$
$$\text{first}_{dmax}(\alpha/\texttt{fol}) = \text{first}_{dmax}(\alpha[\texttt{fol}])/\text{first}_{dmax}(\texttt{fol}) \quad (15)$$
$$\text{first}_{dmin}(\alpha/\texttt{fol}) = \text{first}_{dmax}(\alpha[\texttt{fol}])/\text{first}_{dmin}(\texttt{fol}) \quad (16)$$
$$\text{first}_{dmax}(\alpha/\texttt{par}) = \text{first}_{dmax}(\alpha[\texttt{par}])/\text{first}_{dmax}(\texttt{par}) \quad (17)$$
$$\text{first}_{dmin}(\alpha/\texttt{anc}) = \text{first}_{dmax}(\alpha[\texttt{anc}])/\text{first}_{dmin}(\texttt{anc}) \quad (18)$$
$$\text{first}_{dmin}(\alpha/\texttt{anc}) = \text{first}_{dmin}(\alpha[\texttt{anc}])/\text{first}_{dmin}(\texttt{anc}) \quad (19)$$
$$\text{first}_{dmax}(\alpha/\texttt{desc}) = \text{first}_{dmin}(\alpha[\texttt{desc}])/\text{first}_{dmax}(\texttt{desc}) \quad (20)$$
$$\text{first}_{dmin}(\alpha/\texttt{desc}) = \text{first}_{dmin}(\alpha[\texttt{desc}])/\text{first}_{dmin}(\texttt{desc}) \quad (21)$$
$$\text{last}_{dmax}(\alpha/\texttt{pre}) = \text{last}_{dmin}(\alpha[\texttt{pre}])/\text{last}_{dmax}(\texttt{pre}) \quad (22)$$
$$\text{last}_{dmin}(\alpha/\texttt{pre}) = \text{last}_{dmin}(\alpha[\texttt{pre}])/\text{last}_{dmin}(\texttt{pre}) \quad (23)$$
$$\text{last}_{dmin}(\alpha/\texttt{fol}) = \text{last}_{dmax}(\alpha[\texttt{fol}])/\text{last}_{dmin}(\texttt{fol}) \quad (24)$$
$$\text{last}_{dmin}(\alpha/\texttt{fol}) = \text{last}_{dmin}(\alpha[\texttt{fol}])/\text{last}_{dmin}(\texttt{fol}) \quad (25)$$
$$\text{last}_{dmin}(\alpha/\texttt{par}) = \text{last}_{dmin}(\alpha[\texttt{par}])/\text{last}_{dmin}(\texttt{par}) \quad (26)$$
$$\text{last}_{dmax}(\alpha/\texttt{anc}) = \text{last}_{dmax}(\alpha[\texttt{anc}])/\text{last}_{dmax}(\texttt{anc}) \quad (27)$$
$$\text{last}_{dmax}(\alpha/\texttt{anc}) = \text{last}_{dmin}(\alpha[\texttt{anc}])/\text{last}_{dmax}(\texttt{anc}) \quad (28)$$
$$\text{last}_{dmax}(\alpha/\texttt{desc}) = \text{last}_{dmax}(\alpha[\texttt{desc}])/\text{last}_{dmax}(\texttt{desc}) \quad (29)$$
$$\text{last}_{dmin}(\alpha/\texttt{desc}) = \text{last}_{dmax}(\alpha[\texttt{desc}])/\text{last}_{dmin}(\texttt{desc}) \quad (30)$$

Each application of one of the rules removes a single location step from the argument path. This is repeated until the path length becomes one. The rules given above do not cover all cases, but are the ones that can be easily described in terms of nondeterministic finite automata. (We will see in Section 6 how to treat the other cases.)

The rules indicate which transitions are possible (see Figure 2). The states of the automata are first$_{\texttt{dmax}}$, first$_{\texttt{dmin}}$, last$_{\texttt{dmin}}$, and last$_{\texttt{dmax}}$. These are denoted by $[f|l][\texttt{dmax}|\texttt{dmin}]$. The start states are `fdmax` and `ldmin`. All states are final states. Consequently, the path that occurs as an argument of first$_d$ or last$_d$ can be acceptably rewritten if its reverse is accepted by the automata in Fig. 2.

If the path expression is accepted by the automata, we have a simple way to rewrite the extended path expression such that a first$_d$ or last$_d$ is applied only to a single location step. This eases code generation since we only have to supply code fragments for every combination of these two functions, their possible subscripts `dmin` and `dmax`, and the XPath axes. Together with the code generation of roots($\texttt{desc}/\texttt{child}/\ldots/\texttt{child}$), this is the subject of the next section.

Before proceeding to the next section, let us illustrate how to prove the above by giving an exemplary proof. Treatment of all cases would increase the length of the paper by several dozens of pages. The other proofs follow an analogous scheme and can be found in [12]. We consider the first case (Eqn. 13). Define

$$\begin{aligned} x &:= \text{first}_{dmax}(\alpha/\texttt{pre}) \\ &= \text{first}_{dmax}(\alpha[\texttt{pre}]/\texttt{pre}) \\ &= \text{first}_{dmax}(\alpha[\texttt{pre}]/\text{first}_{dmax}(\texttt{pre})) \\ y &:= \text{first}_{dmax}(\alpha[\texttt{pre}])/\text{first}_{dmax}(\texttt{pre}) \end{aligned}$$

Now assume that $x \neq y$. Since $y \in \alpha[\texttt{pre}]/\text{first}_{dmax}(\texttt{pre})$ we have $x.dmax < y.dmax$. (If they would be equal, we would have $x = y$.) We further have

$$\begin{array}{ll} \exists z_x \in \alpha[\texttt{pre}] & x = z_x/\text{first}_{dmax}(\texttt{pre}) \\ \exists z_y \in \text{first}_{dmax}(\alpha[\texttt{pre}]) & y = z_y/\text{first}_{dmax}(\texttt{pre}) \end{array}$$

From this we can infer that

1. $x.dmin < x.dmax < z_x.dmin < z_x.dmax$
2. $y.dmin < y.dmax < z_y.dmin < z_y.dmax$
3. $z_y.dmax < z_x.dmax$ ('=' $\Longrightarrow \surd$)

From these inequalities we can infer the following. If $x \in z_y/\texttt{pre}$, we must have $x = y$. Otherwise, $y.dmax < z_y.dmin < x.dmax$. Which yields a contradiction.

3.3. Example rewrite

We rewrite the sample path
$/\texttt{desc} :: a/\texttt{anc} :: b/\texttt{desc} :: c/\texttt{fol} :: d/\ \texttt{desc} :: e/\texttt{child} :: f/\texttt{child} :: g$.
First, we introduce the roots step function using Eqn. 6, since the last interesting axis is `desc`. This yields
roots($/\texttt{desc} :: a/\texttt{anc} :: b/\texttt{desc} :: c/\texttt{fol} :: d$)/ $\texttt{desc} :: e/\texttt{child} :: f/\texttt{child} :: g$
The path in roots contains a `fol` axis. Hence, we introduce a first$_{\texttt{dmax}}$ using Eqn 4:
roots(first$_{\texttt{dmax}}$($/\texttt{desc} :: a/\texttt{anc} :: b/\texttt{desc} :: c$)/ $\texttt{fol} :: d$)/$\texttt{desc} :: e/\texttt{child} :: f/\texttt{child} :: g$
The next step is to move first$_{\texttt{dmax}}$ out of roots using Eqn. 3:

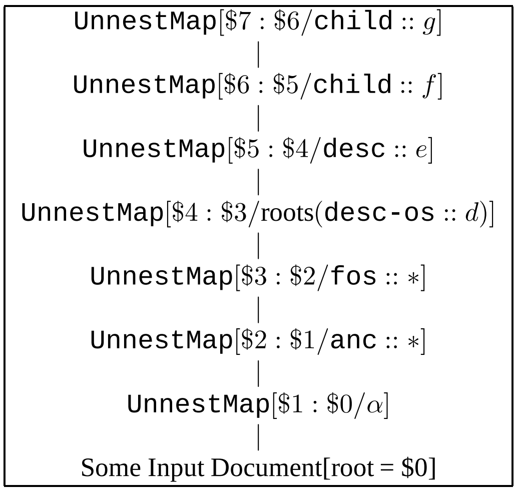

UnnestMap[$7 : $6/child :: g]
UnnestMap[$6 : $5/child :: f]
UnnestMap[$5 : $4/desc :: e]
UnnestMap[$4 : $3/roots(desc-os :: d)]
UnnestMap[$3 : $2/fos :: *]
UnnestMap[$2 : $1/anc :: *]
UnnestMap[$1 : $0/α]
Some Input Document[root = $0]

Figure 3. A Fully Pipelined Plan for the Sample Path

$\text{first}_{\mathtt{dmax}}(/\mathtt{desc} :: a/\mathtt{anc} :: b/\mathtt{desc} :: c)/$
$\text{roots}(\mathtt{fol} :: d)/\mathtt{desc} :: e/\mathtt{child} :: f/\mathtt{child} :: g$
Next, we distribute first over the steps using Eqn. 20 which yields
$\text{first}_{\mathtt{dmin}}(/\mathtt{desc} :: a/\mathtt{anc} :: b[\mathtt{desc} :: c])/$
$\text{first}_{dmax}(\mathtt{desc} :: c)$.
Then we apply Eqn. 19
$\text{first}_{\mathtt{dmin}}(/\mathtt{desc} :: a[\mathtt{anc} :: b[\mathtt{desc} :: c]])/$
$\text{first}_{dmin}(\mathtt{anc} :: b[\mathtt{desc} :: c])/\text{first}_{dmax}(\mathtt{desc} :: c)$
Call this expression α. We expand the `fol` axis:
$\text{roots}(\mathtt{anc} :: */\mathtt{fos} :: */\mathtt{desc\text{-}os} :: d)/\mathtt{desc} :: e/$
$\mathtt{child} :: f/\mathtt{child} :: g$
which is equivalent to:
$\mathtt{anc} :: */\mathtt{fos} :: */\text{roots}(\mathtt{desc\text{-}os} :: d)/\mathtt{desc} :: e/$
$\mathtt{child} :: f/\mathtt{child} :: g$
A sequence of interleaved first sequences recognized by the nested predicates (as in α) is translated into a single `UnnestMap`. Its programs avoid duplicate evaluation of those paths that are contained in the predicates as well as in the subsequent top-level path. The plan for our example path is shown in Fig. 3. Note, that this plan is fully pipelined, α can be evaluated very efficiently and returns only a single node, and no duplicates are produced in any step.

The next section will show how to translate the subscripts of the plan into `init` and `step` programs. Note that the top three `UnnestMap` operations can be handled together to make the program more efficient.

4. Code Generation

As we have seen in the last section, two important patterns emerged for which we want to be able to generate efficient code. One is an `UnnestMap` with a sequence of interleaved first_d (or last_d) sequences. The other is an `UnnestMap` for a root step operating on a descendant axis followed by several child axes.

We generate code for the operations `open` (called `init` in our case) and `next` (called `step` in our case) of an iterator allowing pipelined processing. The program for `init` initializes an iterator and hands back the first qualifying node. Each call of `next` returns the next qualifying node.

We start with the code generation for a sequence of first_d (last_d steps are handled analogously). Then we consider the example programs for handling a first_{dmin} with a `descendant` axis. A complete list of programs can be found in [12].

Input to the code generation is the current context node `cn[0]` and a sequence of steps (with the individual steps stored in an array `step[]` and the corresponding predicates in an array `pred[]`).

The steps of a sequence are processed from left to right, so we do not know yet for an intermediate node, if, after applying the remaining steps, any qualifying nodes exist. That means, that during evaluation of a sequence we may have to backtrack. For this reason we use two subroutines in our code generation algorithm: one to find the first node for a step, called `mini-init` and another one to find subsequent nodes, called `mini-step`.

The subroutine `mini-init(cn, step, pred)` has as its output the first qualifying node for a given context node, a first_d step, and its predicates. The subroutine `mini-step(cn_orig, cn_current, step, pred)` navigates to the next qualifying node given the original context node from `mini-init`, a (qualifying) starting node, a first_d step and its predicates (again excluding predicates concerning later steps).

Let us now present the init part of the algorithm for the code generation of first_d steps:

```
init:
  cn[1] = mini-init(cn[0], step[1],
                    pred[1]);
label1:
  if(cn[1] == NULL) { return NULL; }
  cn[2] = mini-init(cn[1], step[2],
                    pred[2]);
label2:
  if(cn[2] == NULL) {
    cn[1] = mini-step(cn[0], cn[1],
                      step[1], pred[1]);
    goto label1;
  }
  ...
  cn[n] = mini-init(cn[n-1], step[n],
                    pred[n]);
labeln:
  if(cn[n] == NULL) {
    cn[n-1] = mini-step(cn[n-2], cn[n-1],
                   step[n-1], pred[n-1]);
    goto labeln-1;
  }
  return cn[n];
```

The `UnnestMap` operator for a sequence of first$_d$ steps returns at most one qualifying node. So the step program is very simple, it always returns NULL, as the lone potential node is already returned by the init program.

In order to get a feel for the `mini-init` and `mini-step` programs, we present the programs for a first$_{dmin}$ step with a `descendant` axis.

For the `mini-init` program we just have to step through the descendants of the current context node in preorder. The first node qualifying the predicates is returned.

```
mini-init(cn, first_dmin(desc), pred) {
mini-init:  start = cn;
            if(nextpreorder(start) does
               not exist) return NULL;
            next = nextpreorder(start);
label:      if(next is not descendant
               of start) return NULL;
            if(next satisfies pred)
              return next;
            if(nextpreorder(next) does
               not exist) return NULL;
            next = nextpreorder(next);
            goto label;
          }
}
```

The corresponding step program traverses the subsequent qualifying nodes in preorder.

```
mini-step(cn_orig, cn_current,
          first_dmin(desc), pred) {
mini-step:  start = cn_current;
            if(nextpreorder(start) does
               not exist) {
              return NULL;
            }
            next = nextpreorder(start);
label:      if(next is not descendant
               of cn_orig) {
              return NULL;
            }
            if(next satisfies pred) {
              return next;
            }
            if(nextpreorder(next) does
               not exist) {
              return NULL;
            }
            next = nextpreorder(next);
            goto label;
          }
}
```

Note that the two procedures for the init and step program are very similar. However, since our `UnnestMap` operator takes these programs as different parameters, we don't generalize these programs to a single one.

5. Evaluation

In this section we present a first round of experiments, which were performed to validate the effectiveness of our technique.

The experimental environment was an Intel Pentium II with 400 MHz, running Linux 2.4. The tests used Natix' query execution engine, the Natix Virtual Machine (NVM), written in C++ and compiled with optimizing gcc 3.1. As a reference, the same XPath expressions were also evaluated with the Xalan XSLT processor, also compiled with optimizing gcc, and using a very simple stylesheet which only selects the nodes reachable by the given XPath expression, doing nothing with them.

We evaluated four different XPath expressions, using the same three XML documents each time, and starting from the document root node as context node. The documents were of a simple recursive structure with a fixed fanout on each inner node, with a constant tree height of five. All the nodes carried the same tag name, and the leaf nodes were empty tags with no text.

The results are shown in Figure 4. The measurements of evaluation times were averaged over a series of program runs and do not include the overhead caused by program startup. These overhead times are shown at the bottom of the table. For Xalan, they grow with the document size because the document must be parsed first, while in Natix it is already stored as a tree structure.

The queries are shown in the table as a sequence of axes, using the abbreviations from Sec. 2. The node test used for each location step is always the same tag name (the one from the document).

The first expression is only shown as a reference. It computes all `descendant`s of the document root. No intermediate results are generated. Xalan is faster than a simple NVM `UnnestMap` because NVM operates directly on the secondary storage structure while Xalan navigates using

		Fanout		
XPath	Method	4	5	6
`desc`	NVM	0.0104	0.0146	0.0654
	Xalan	0.0036	0.0112	0.0232
`desc/desc`	NVM DupElim	0.0520	0.1146	0.3032
	NVM Pipe	0.0076	0.0298	0.0868
	Xalan	0.0196	0.0600	0.1424
`desc/fol`	NVM DupElim	7.9492	53.0264	332.0924
	NVM Pipe	0.0192	0.0438	0.1278
	Xalan	2.9752	25.3396	155.3674
`desc/fol/desc`	NVM DupElim	27.3152	201.6034	1252.0924
	NVM Push	8.0452	53.5834	353.9734
	NVM Pipe	0.0192	0.0492	0.1332
	Xalan	14.2622	130.0896	717.2614
Startup	NVM	0.0848	0.0866	0.0866
Startup/Parsing	Xalan	0.2278	0.4204	0.8386
Document Size		20K	58K	140K

Figure 4. Experimental results (time in seconds)

main memory pointers, and the effort for parsing the document in Xalan is not included in the numbers.

The second expression consists of two `descendant` axes. The three evaluation methods shown are NVM using a final duplicate elimination operator to eliminate duplicates (NVM DupElim), using a pipelined plan as explained in sections 3 and 4 (NVM Pipe), and using Xalan. The pipelined execution outperforms the other evaluation methods by factors up to 2.5.

The third expression combines a `descendant` with a `following` axis, resulting in the generation of a large repeating node sequence from the `following` step in the conventional methods (NVM DupElim and Xalan), which is subject to duplicate elimination. The pipelined version (NVM Pipe) still exhibits performance figures close to a single tree traversal.

In the last path expression, the third XPath expression was extended by another `descendant` axis. We used two approaches without pipelining here, differing in the point of time when the duplicates are eliminated. We can either use only a final duplicate elimination operator (NVM DupElim), or push another duplicate elimination into the execution plan after the first two axes have been evaluated (NVM Push). Pushing a second duplicate elimination is already an improvement, reducing the number of intermediate nodes considerably. This brings the execution time of the last expression down close to the execution time of the third expression with a single duplicate elimination. Xalan's performance figures indicate that it does not perform an intermediate duplicate elimination (as also pointed out in [9]). The pipelined execution performance is still in the vicinity of a single tree traversal, making it several thousand times faster than NVM DupElim, NVM Push and Xalan.

Summarizing, the first experimental evaluation of pipelined XPath is very promising. It is our intention to verify the robustness of our technique against different selectivities of node tests, and against different classes of document structure.

6. Extensions

Let us reexamine the restrictions mentioned in the introduction. Out of our equivalences, only Eqns. 4 to 7 pose any problems for occurrences of `position()` or `last()`. They do so, if the predicate of the axis in S that is used as the anchor point contains one of these functions. It is easy to verify that any other occurrence does not pose any problems.

The other restrictions have to do with the rewriting step functions. The automata in Fig. 2 do not accept the full XPath language. We now show how additional transitions can be added to rectify this situation. As an example let us consider the `child` axis. Assume we want to compute $\text{first}_{dmin}(\alpha/\texttt{child})$. We define $x := \text{first}_{dmin}(\alpha/\texttt{child})$,
$y := \text{first}_{dmin}(\alpha[\texttt{child}])/\text{first}_{dmin}(\texttt{child})$, and z_x and z_y such that $x = z_x/\text{first}_{dmin}(\texttt{child})$ and $y = z_y/\text{first}_{dmin}(\texttt{child})$. It is then easy to show that (see [12]) either $x = y$ or

$$z_y.dmin < z_x.dmin < x.dmin < x.dmax < z_x.dmax < y.dmin < y.dmax < z_y.dmin$$

where $z_x = y$ is possible.

Let us interpret this result. Whenever we want to compute x, we can compute y. This gives us some nodes z_y and y. Either $y = x$ and we computed the correct node, or there exists a node z_x somewhere in $z_y/\texttt{desc}$ whose child x is the correct solution. Hence, we can start over and compute some solution z'_y, y underneath z_y. We iterate this procedure until no further qualifying nodes can be found in the descendants of our current solution. It is easy to extend code generation procedure to take care of this fact. Similar arguments apply to the `fos` and `prs` axes. Hence, these axes can be treated within our approach.

To complete the automata to accept the full XPath language, in some cases it is not sufficient to repeatedly search downwards within the descendants of a node. Sometimes we have to repeatedly check the ancestors of an intermediate solution. We can handle the complete XPath language by extending our code generation appropriately. For details see [12].

The only axis that still poses problems is the `ancestor` axis. Currently, we do not have an elegant solution within our framework. Hence, we plan to implement a special algebraic operator that computes all ancestors for a given set of nodes and concurrently performs duplicate elimination.

7. Conclusion

We presented an approach for the efficient evaluation of XPath expressions by pipelining the individual location steps. This is not straightforward, as several difficulties have to be overcome. Among these are duplicate elimination and cutting down massively on the number of nodes visited. We reach our goal via extensive transformations of the original XPath expression allowing uncomplicated code generation.

In addition to theoretical work, we implemented a prototype of our approach in our native XML database system Natix. The conducted experiments demonstrate the dominance of our approach in terms of performance. We were always faster than other methods, typically up to several orders of magnitude.

We are optimistic that the few points remaining open can be settled in a satisfying manner and that pipelining is the right approach for efficiently processing XPath expressions.

Acknowledgment We thank Simone Seeger for her help in preparing the manuscript and the anonymous referees for their helpful comments.

References

[1] S. Al-Khalifa, H. Jagadish, N. Koudas, J. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proc. IEEE Conference on Data Engineering*, pages 141–152, 2002.

[2] M. Altinel and M. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 53–64, 2000.

[3] S. Amer-Yahia, S. Cho, L. Lakshmanan, and D. Srivastava. Efficient algorithms for minimizing tree pattern queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 497–508, 2001.

[4] F. Bry, D. Olteanu, H. Meuss, and T. Furche. Symmetry in XPath. Technical Report PMS-FB-2001-16, LMU, München, 2001.

[5] C. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath. In *Proc. IEEE Conference on Data Engineering*, pages 235–244, 2002.

[6] C. Clarke, G. Cormack, and F. Burkowski. An algebra for structured text search and a framework for its implementation. *The Computer Journal*, 38(1):43–56, 1995.

[7] T. Fiebig and G. Moerkotte. Evaluating Queries on Structure with eXtended Access Support Relations. In D. Suciu and G. Vossen, editors, *The World Wide Web and Databases*, LNCS 1997, pages 125–136. Springer, 2001.

[8] T. Fiebig and G. Moerkotte. Algebraic XML construction and its optimization in Natix. *World Wide Web Journal*, 4(3):167–187, 2002.

[9] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 95–106, 2002.

[10] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2), June 1993.

[11] T. Grust. Accelerating XPath location steps. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 109–120, 2002.

[12] S. Helmer, C.-C. Kanne, and G. Moerkotte. Optimized translation of XPath expressions into algebraic expressions parameterized by programs containing navigational primitives. Technical Report 11, University of Mannheim, 2002.

[13] C.-C. Kanne and G. Moerkotte. Efficient storage of XML data. In *Proc. IEEE Conference on Data Engineering*, page 198, 2000.

[14] A. Kwong and M. Gertz. Schema-based optimization of XPath expressions. Technical report, UC Davis, 2002.

[15] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *Workshop on XML-Based Data Management (XMLDM) in conjunction with EDBT*, 2002.

[16] P. Ramanan. Efficient algorithms for minimizing tree pattern queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 299–309, 2002.

[17] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 204–215, 2002.