Full-fledged Algebraic XPath Processing in Natix

Matthias Brantner Sven Helmer Carl-Christian Kanne Guido Moerkotte
Universität Mannheim
Mannheim, Germany
msb|helmer|cc|moer@pi3.informatik.uni-mannheim.de

Abstract

We present the first complete translation of XPath into an algebra, paving the way for a comprehensive, state-of-the-art XPath (and later on, XQuery) compiler based on algebraic optimization techniques. Our translation includes all XPath features such as nested expressions, position-based predicates and node-set functions.

The translated algebraic expressions can be executed using the proven, scalable, iterator-based approach, as we demonstrate in form of a corresponding physical algebra in our native XML DBMS Natix. A first glance at performance results shows that even without further optimization of the expressions, we provide a competitive evaluation technique for XPath queries.

1. Introduction

The efficient processing of XML data hinges on fast evaluation techniques for XPath expressions, because XPath is an essential part of widely used XML processing languages like XSLT and XQuery. We present the first complete translation of XPath into an algebra.

Such a translation of XPath expressions into algebraic expressions (1) renders possible algebraic optimization approaches as found in most modern query optimizers, and (2) facilitates the application of iterator-based, pipelining query execution engines that scale well to large data volumes and have proven their performance e.g. in relational systems. For the same reasons, algebra-based XQuery evaluation is attractive, requiring algebra-based XPath evaluation as an essential ingredient.

The main contributions of our paper are:

- We introduce an algebra capable of expressing *any* XPath query
- We show exactly *how* all XPath constructs can be translated into algebraic expressions

These contributions are not intended to be purely theoretical exercises. To show their usefulness in implementing XPath evaluators, we also discuss our compiler and algebra implementation, and give some performance results.

The current approaches for evaluating XPath (e.g. [3, 5, 7, 8, 11, 14, 19, 21, 22, 24]) can be divided into several different categories. First of all, we have (main-memory-based) interpreters [19, 22]. Although most of them support the full XPath standard, they have high memory requirements and do not scale to large documents very well. Second, many papers were published investigating the efficient evaluation of individual location steps [3, 14, 24]. For some location steps very efficient operators have been developed, but a complete framework for supporting the full XPath standard seems still to be missing, e.g. there is no support for nested expressions or position-based predicates. Third, we have approaches relying on relational databases [7, 11, 12]. Here, the XML data is transformed and stored in relations. Queries containing XPath expressions are translated into SQL and processed using the (possibly extended) engine of the underlying database system. Finally, there are other algebra-based approaches for evaluation of queries over XML data [1, 16, 23], However, these approaches give no translation function for all of XPath's constructs, in particular the whole set of axes.

In our approach we translate XPath 1.0 expressions[1] into a logical algebra working on *ordered tuple sequences*. The main task here is avoiding unnecessary work by *eliminating duplicates* in intermediate results or memoizing already computed results (such as location steps or predicates) if duplicate elimination is not possible. This is very important, as the presence of duplicates may lead to an exponential run time [8, 9]. Another important point we cover is the efficient evaluation of predicates in XPath. We pay particular attention to *position-based predicates* using *position()* or *last()*. The bottom-up approach in [9] computes all potential contexts to do this, thus performing unnecessary computations. Their top-down approach is not algebraic and re-

1 We will only write XPath in the following, always meaning XPath 1.0 except when explicitly stated otherwise.

quires materialization of all intermediate results, which we circumvent using our algebraic approach.

For query execution, we use the physical algebra of our native XML database system Natix [6], which implements the operators of the logical algebra in an iterator-based fashion [10]. We do not need to construct a complete main memory representation of an XML document in order to evaluate XPath expressions. Our approach directly accesses the physical storage layout of the XML documents on disk. Finally, our XPath evaluator is implemented in a modular way, allowing the integration of several different optimization techniques.

The remainder of this paper is organized as follows: In Sec. 2, we summarize the XPath semantics and introduce our logical algebra. Sec. 3 describes the canonical translation of XPath expressions into our algebra, and Sec. 4 shows how to avoid exponential run-time of the queries. Implementation details concerning our physical algebra are given in Sec. 5. Space constraints prohibit to incorporate XPath specific algebraic optimization techniques in this paper. However, in Sec. 6 preliminary performance results show that even without further optimization, our approach compares favorably to main-memory based evaluators, and scales better to large document sizes. Sec. 7 summarizes our contributions and outlines future work.

2. Translation Input and Output

This section explains domain and range of our translation function: We give a brief summary of XPath expression semantics, and introduce our logical algebra.

2.1. XPath Semantics

The primary syntactic construct in XPath is an expression. When evaluating an expression, the result has one of the following four basic types: a node set (an unordered collection of nodes without duplicates), a boolean value ('true' or 'false'), a number (a floating-point number), or a string (a sequence of characters). The evaluation of an expression considers a context, which consists of the following: a node (also called the context node), a pair of nonnegative integers (the context position and context size), a set of variable bindings (a map from variable names to values), a function library, and a set of namespace declarations.

Please note that in XPath 1.0, the node sets themselves are unordered. However, there exists the notion of document order, totally ordering all nodes of a document. Document order is relevant in the evaluation of location steps, but not in the representation of node sets. Hence, we do not always return result sequences in document order. For XPath 2.0 (and integration into XQuery), if ordered results are required, additional sorting is sometimes [15] necessary.

2.2. Logical Algebra

Before going into the details of the translation we have to define the target algebra and some associated notions.

2.2.1. Universe The universe of our algebra is the union of the domains of the atomic XPath types (`string, number, boolean`) and the set of ordered sequences of tuples[2].

A tuple is a mapping from a set of attributes to values. We allow nested tuples, i.e. the value of an attribute may be a sequence of tuples. In addition to sequences, attribute values may be document nodes or values of the atomic XPath types.

2.2.2. Conventions Before defining the main algebra operators below, we introduce the notations used in their definition and in the description of the translation process:

The set of attributes defined for a tuple t is written as $\mathcal{A}(t)$. All the tuples $t \in e$ of a sequence-valued expression e have the same attributes $\mathcal{A}(t)$, which are also denoted $\mathcal{A}(e)$. The set of free variables of an expression e is defined as $\mathcal{F}(e)$.

Single tuples are constructed by using the standard $[\cdot]$ brackets. The concatenation of tuples and functions is denoted by $\circ$.

The projection of a tuple on a set of attributes A is denoted by $|_A$. We also define $t|_{\overline{A}} := t|_{A(t)\bar{A}}$. For brevity reasons, we identify a tuple containing a single attribute with the value of that attribute.

For an expression e possibly containing free variables, and a tuple t, we denote by $e(t)$ the result of evaluating e where bindings of free variables are taken from attribute bindings provided by t. Of course this requires $\mathcal{F}(e) \subseteq \mathcal{A}(t)$. In general, accesses to identifiers are resolved by lookup in the tuple; if no mapping can be found, the tuples of the surrounding algebra expressions are checked successively. Ultimately, the free variables of the complete expressions must be bound by a top-level map supplied as execution context for the expressions. This top-level map also must provide bindings for the XPath $ variables and the context node for the execution.

For sequences e we use $\alpha(e)$ to denote the first element of a sequence. We identify single element sequences and elements. The function τ retrieves the tail of a sequence and $\oplus$ concatenates two sequences. We denote the empty sequence by ϵ. As a first application, we construct from a sequence of non-tuple values e a sequence of tuples denoted by $e[a]$. It is empty if e is empty. Otherwise $e[a] = [a : \alpha(e)] \oplus \tau(e)[a]$.

By *id* we denote the identity function.

2 We use ordered sequences instead of node-sets, since predicate evaluation (with $position()$ and $last()$), and embedding of XPath into other languages is sensitive to the order of the returned nodes.

Selection σ	selects qualifying tuples according to predicate p: $\sigma_p(e) := \begin{cases} \alpha(e) \oplus \sigma_p(\tau(e)) & \text{if } p(\alpha(e)) \\ \sigma_p(\tau(e)) & \text{else} \end{cases}$
Projection $\mathbf{\Pi}$	projects on attributes in A (duplicate elimination version called Π^D_A, duplicate elimination without projection denoted by Π^D, attribute renaming version denoted by $\Pi_{a':a}$): $\Pi_A(e) := \alpha(e)\|_A \oplus \Pi_A(\tau(e))$ $\Pi_{a':a}(e) := \alpha(e)\|_a \circ [a' : a] \oplus \Pi_A(\tau(e))$
Map χ	extends each tuple t_i in e_1 with attribute a with value of $e_2(t_i)$: $\chi_{a:e_2}(e_1) := \alpha(e_1)\|_{Attr(e_1)\setminus\{a\}} \circ [a : e_2(\alpha(e_1))] \oplus \chi_{a:e_2}(\tau(e_1))$
Product $\overline{\times}$	connects single tuple t_1 to each tuple in e_2: $t_1 \overline{\times} e_2 := (t_1 \circ \alpha(e_2)) \oplus (t_1 \overline{\times} \tau(e_2))$
Cross product $\times$	connects all tuples in e_1 to all in e_2: $e_1 \times e_2 := (\alpha(e_1) \overline{\times} e_2) \oplus (\tau(e_1) \times e_2)$
D-join $<>, \underrightarrow{\bowtie}$	joins each tuple t_i in e_1 to all tuples in e_2, which depend on t_i: $e_1 <e_2> := \alpha(e_1) \overline{\times} e_2(e_1) \oplus \tau(e_1) <e_2>$
Semi-join $\ltimes$	p checks for tuple existence in e_2 to decide on including tuple in e_1: $e_1 \ltimes_p e_2 := \begin{cases} \alpha(e_1) \oplus (\tau(e_1) \ltimes_p e_2) & \text{if } \exists x \in e_2 \;\; p(\alpha(e_1) \circ x) \\ \tau(e_1) \ltimes_p e_2 & \text{else} \end{cases}$
Anti-join $\triangleright$	p checks for tuple non-existence in e_2 to decide on including tuple in e_1: $e_1 \triangleright_p e_2 := \begin{cases} \alpha(e_1) \oplus (\tau(e_1) \triangleright_p e_2) & \text{if } \nexists x \in e_2 \;\; p(\alpha(e_1) \circ x) \\ (\tau(e_1) \triangleright_p e_2) & \text{else} \end{cases}$
Unnesting μ	unnests a sequence-valued nested attribute: $\mu_g(e) := (\alpha(e)\|_{\overline{\{g\}}} \times \alpha(e).g) \oplus \mu_g(\tau(e))$
Unnest-Map $\mathbf{\Upsilon}$	abbreviated notation for a map operator followed by an unnest operator: $\Upsilon_{a:e_2}(e_1) := \mu_g(\chi_{g:e_2[a]}(e_1))$
Binary Grouping $\mathbf{\Gamma}$	Adds to e_1 an attribute based on aggregation of e_2 $G(x) := f(\sigma_{x\|_{A_1} \theta A_2}(e_2))$: $e_1 \Gamma_{g;A_1 \theta A_2;f} e_2 := \alpha(e_1) \circ [g : G(\alpha(e_1))] \oplus (\tau(e_1) \Gamma_{g;A_1 \theta A_2;f} e_2)$
Aggregation $\mathfrak{A}$	Aggregates input sequence into a singleton sequence with a single attribute a: $\mathfrak{A}_{a;f}(e) := \{[a : f(e)]\}$
Sorting $\mathbf{S}ort$	Sorts input sequence based on attribute a: $Sort_a(e) := Sort_a(\sigma_{a<\alpha(e).a}(\tau(e))) \oplus \alpha(e) \oplus Sort_a(\sigma_{a\geq\alpha(e).a}(\tau(e)))$
Singleton Scan $\square$	Returns singleton sequence consisting of the empty tuple: $\square := \{[\,]\}$

Figure 1. Sequence-valued operators of the target algebra

2.2.3. Operators The main operators of our algebra are sequence-valued analogs of traditional database algebra operators. An overview of the formal definitions of the sequence-valued operators is given in Fig. 1. More detailed comments about the operators and their usage is embedded in the description of our translation process in the remainder of the paper.

Except if explicitly stated otherwise, unary operators produce ϵ if their input is ϵ, and binary operators produce ϵ if their left input is ϵ. The d-join has two notations, one to be used in visualizations of query trees ($\underrightarrow{\bowtie}$) which designates the dependent side using an arrow, and one for textual expressions where its dependent side is parenthesized ($<>$).

In addition, our target algebra provides counterparts for functions (e.g. `contains`) and operators (e.g. $+, *, /, =$) defined on the XPath basic types, including explicit and implicit conversions. For those functions that have node-sets as inputs (e.g. `count`), their algebraic counterpart has sequence-valued input. Note that for some XPath functions and operators, special translation rules are given in Sec. 3 (in particular node-set comparison, see Sec. 3.6). These functions or operators have no direct equivalent in our algebra.

3. Translation into Algebra

In a first translation step we decide for each expression a mapping onto algebraic operators. In a second step we enhance the translation to avoid exponential complexity of the evaluation process. The description of our translation pro-

cess follows loosely the XPath grammar as found in the W3C recommendation [4].

When translating XPath into our algebra we denote the translation of an expression e by $\mathcal{T}[e]$. The result of our translation function is an algebraic expression which may or may not be sequence-valued.

3.1. Location Paths

The most important construct in XPath is a location path. Location paths are applied to context nodes and produce as a result a node set (Sec. 2.1).

We have to distinguish between absolute and relative location paths. An *absolute path* starts at the root element of an XML document. A *relative path* can start at an arbitrary context node. After that, both location paths are handled in the same manner.

The starting context node for a location path is provided by the variable cn. Note that for top-level location paths, cn is free and must be bound by the execution context; this is the mechanism for the execution engine to provide the initial context node.

3.1.1. Canonical Translation A path expression $\pi = \pi_1/s_2/\ldots/s_{n-1}/s_n$ consists of a number of locations steps (denoted by s_i). For the moment, we assume that π starts with a partial expression π_1, consisting of the first location step of π, possibly prefixed by an initial $/$. We take a closer look at π_1 when distinguishing absolute from relative paths below. The individual steps are evaluated sequentially, i.e., the output of a location step s_i serves as the context for the following step s_{i+1}.

We translate a path expression into a chain of dependency joins (d-joins). In a d-join the free variables in the expression on the right-hand side are bound with values supplied from a tuple generated by the expression on the left hand side. We use this mechanism of a d-join to hand over the context from one location step to the next, one node at a time. The independent (left) subexpression of the d-join enumerates the context nodes from the previous step. The dependent subexpression of the d-join has the current step's context node as a free variable. Hence, each evaluation of the dependent subexpression corresponds to one result context of the location step.

We call a translation into d-joins the *canonical translation* of π:

$$\mathcal{T}[\pi] := \Pi^D(\chi_{cn:c_n}(\mathcal{T}[\pi_1] < \mathcal{T}[s_2] > \ldots < \mathcal{T}[s_n] >))$$

We always want the cn attribute in a tuple sequence to contain the node attribute that was last added to the tuple. This makes it easy to treat all sequence-valued algebraic expressions uniformly. That is why we also add a map operator mapping the result nodes of the last step to cn. The precise origin of the c_n attribute is explained below when we elaborate on location steps.

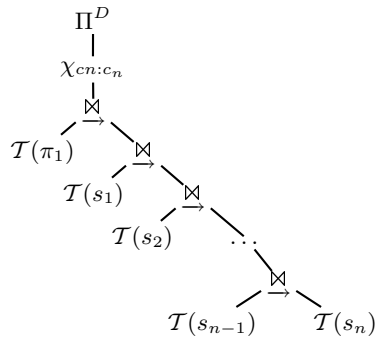

Figure 2. canonical translation

We also have to add a projection operator that eliminates duplicates, as by definition the result of an XPath expression may not contain any duplicates (see also 2.1). The duplicate elimination only operates on the relevant context node attribute cn of the tuple, without projecting away the remaining attributes.

A graphical representation of the translated expression is shown in Fig. 2.

3.1.2. Absolute and relative paths The initial context of a location path depends on whether it is an absolute or relative path, i.e. whether π_1 is prefixed by a slash or not. The translation takes this into account by parameterizing a map operator differently. The map operator supplies the input context node c_0 for the first location step s_1:

$$\mathcal{T}[\pi] := \Pi^D(\chi_{cn:c_n}(\chi_{c_0:c}(\Box) < \mathcal{T}[s_1] > \ldots < \mathcal{T}[s_n] >))$$

with $c = root(cn)$ for an absolute path and $c = cn$ for a relative path. cn must be bound to the context in which to evaluate π.

3.1.3. Unions The union of path expressions $(\pi_1|\pi_2|\ldots|\pi_n)$ is translated into a series of concatenation operators followed by a duplicate elimination:

$$\mathcal{T}[\pi_1|\pi_2|\ldots|\pi_n] := \Pi^D(\mathcal{T}[\pi_1] \oplus \mathcal{T}[\pi_2] \oplus \ldots \oplus \mathcal{T}[\pi_n])$$

Note that the translation of the π_i already binds cn to the produced context node, so no extra map or projection is required.

3.2. Location Steps

A location step consists of three parts: an axis (which specifies the relationship between the result set of nodes and the context node), a node test (which specifies the node

type and name of the selected nodes), and an arbitrary number of predicates (which use additional expressions to further refine the set of selected nodes). We will look at predicates in more detail in the following section, here we address axes and node tests. So for the moment a step s_i is defined by an axis a_i and a node test t_i.

We translate the evaluation of a location step into an unnest map operator, which generates for each input node (represented by a tuple) a sequence of nodes (also represented by tuples) that are reachable from the input node by the specified axis, and satisfy the node test. We explain in Sec. 5.2 how evaluation of the subscript is performed, here it is sufficient to note that the result sequence is in the proper order for the specified axis. The result nodes of step s_i are assigned to the attribute c_i.

$$\mathcal{T}[a_i :: t_i] := \Upsilon_{c_i:(c_{i-1}/a_i::t_i)}(\Box)$$

For two neighboring location steps, e.g. in the location path $a_1 :: t_1/a_2 :: t_2$, it can be seen quite nicely that the result of the location step $a_1 :: t_1$, which is stored in the attribute c_1, is used during the evaluation of location step $a_2 :: t_2$:

$$\Pi^D(\chi_{cn:c_2}(\chi_{c_0:cn}(\Box) < \Upsilon_{c_1:(c_0/a_1::t_1)}(\Box) >< \Upsilon_{c_2:(c_1/a_2::t_2)}(\Box) >))$$

3.3. Predicates

A location step s_i may contain an arbitrary number h of predicates p_k and has the general form $a_i :: t_i[p_1] \dots [p_h]$. The pattern for translating a location step $a_i :: t_i[p_1] \dots [p_h]$ with predicates is

$$\Phi[p_h] \circ \dots \circ \Phi[p_1] \circ \Upsilon_{c_i:c_{i-1}/a_i::t_i}(\Box)$$

where Φ is an auxiliary translation function for predicates, returning a filtering functor which operates on algebraic expressions. We now elaborate on Φ:

An individual predicate p_k is represented as the conjunction of several clauses l_{kj}, i.e., $p_k = \bigwedge_{j=1}^{m_k} l_{kj}$. Depending on whether the conjuncts contain nested location paths, or function calls to the position-based functions $position()$, and $last()$, we have to translate them differently.

3.3.1. Simple clauses Clauses with no positional predicates or nested paths are easiest to translate. Translating a predicate $p_k = l_{k1} \wedge \dots \wedge l_{km_k}$ that does not include positional clauses simply results in a translation into selection operators:

$$\Phi[l_{k1} \wedge \dots \wedge l_{km_k}] := \sigma_{\mathcal{T}[l_{km_k}]} \circ \dots \circ \sigma_{\mathcal{T}[l_{k1}]}$$

After the semantic analysis all clauses are broken down into function calls: $l_{kj} = f_1 \circ \dots \circ f_r$. For example, `or`, `not`, and comparisons are all evaluated by function calls. All implicit conversions have also been added as function calls. We cover the translation of these calls in Sec. 3.6.

3.3.2. Nested paths If any of the l_{kj} contain nested paths, their correct translation requires the cn variable to be bound to their starting context node. Hence, if translating predicates with nested paths, we need to rebind the cn variable to the current context node:

$$\Phi[l_{k1} \wedge \dots \wedge l_{km_k}] := \sigma_{\mathcal{T}[l_{km_k}]} \circ \dots \circ \sigma_{\mathcal{T}[l_{k1}]} \cdots \circ \chi_{cn:c_i}$$

If a nested path is not used inside an aggregate function, the translation will add a conversion to boolean in form of our internal $exists()$ aggregate function (Sec. 3.6).

3.3.3. Clauses with $position()$ If at least one of the clauses in p_k contains $position()$ (but none of them contains $last()$), we have to count the number of context nodes that are produced. We do this with the help of a map operator that labels the tuples of the resulting nodes with their appropriate position within the current context (introducing a new attribute `cp`):

$$\Phi[l_{k1} \wedge \dots \wedge l_{km_k}] := \sigma_{\mathcal{T}[l_{km_k}]} \circ \dots \circ \sigma_{\mathcal{T}[l_{k1}]} \circ \chi_{cp:counter(p_k)++}$$

Calls to $position()$ are then translated into attribute accesses to cp:

$$\mathcal{T}[position()] := cp$$

3.3.4. Clauses with $last()$ The most difficult case are clauses that contain $last()$. Here we have to compute the context size to be able to evaluate the clause. We do this with the help of our new Tmp^{cs} operator that first materializes the context and then adds a context size attribute cs to all the tuples belonging to the current context. In the canonical translation, the context is exactly the result of the current location step's dependent subexpression. Hence, on a logical level Tmp^{cs} is just shorthand for[3]

$$Tmp^{cs}(e) := \mathfrak{A}_{cs;count}(e) \overline{\times} e$$

This leads to the translation of a predicate relying on full positional information as

$$\begin{aligned}\Phi[l_{k1} \wedge \dots \wedge l_{km_k}] \quad := \quad & \sigma_{\mathcal{T}[l_{km_k}]} \circ \dots \circ \sigma_{\mathcal{T}[l_{k1}]} \\ & \circ \mathrm{Tmp}^{cs} \circ \chi_{cp:counter(p_k)++}\end{aligned}$$

with

$$\mathcal{T}[last()] := cs$$

3.4. Filter Expressions

XPath allows to filter any expression of type node-set using predicates. As with location path predicates, we use a different translation in case there are position-based clauses.

3 We explain in Sec. 5.2 how to implement Tmp^{cs} efficiently.

3.4.1. Without position-based predicates If the predicates p_i in the filter expression $e[p_1]\dots[p_h]$ do not contain $position()$ or $last()$, we have as translation:

$$\mathcal{T}\,[e[p_1]\dots[p_h]] := \Phi[p_h] \circ \dots \circ \Phi[p_1] \circ \mathcal{T}[e]$$

Note that the sequence-valued e already has cn bound to the correct node, so we do not need to add a map operator.

3.4.2. With position-based predicates Position-based predicates in filter expressions are evaluated with respect to the child axis, i.e. in document order. In location step predicates, the input sequence (the context) always results from a single location step and hence is properly ordered. In filter expressions, the input sequence may contain an arbitrary node sequence. To make the counting mechanisms from Sec. 3.3 work for filter expressions, we must guarantee that the input sequence is in document order. Hence, we introduce a sort operator which establishes document order before evaluating the predicates[4]. So, if there is any predicate p_k in the filter expression $e[p_1]\dots[p_h]$ containing $position()$ or $last()$, its translation is

$$\mathcal{T}\,[e[p_1]\dots[p_h]] := \Phi[p_h] \circ \dots \circ \Phi[p_1] \circ Sort_{cn}(\mathcal{T}[e])$$

3.5. Path Expressions

Path Expressions are a more general form of relative location paths. They comprise a node-set expression e and a relative location path π. All the nodes in the node set are used as context nodes for the location path, and a union of the results is returned.

Our translation of path expressions uses a d-join to feed all nodes from e as context nodes to the relative location path:

$$\mathcal{T}[e/\pi] := \Pi^D(\Pi_{cn:c'}(\mathcal{T}[e] < \Pi_{c':cn}(\mathcal{T}[\pi]) >))$$

The duplicate elimination operator is required since the evaluation of π for several context nodes may introduce duplicates, just as in location paths.

Note that the tuple sequence from e has an attribute cn containing the nodes it provides. The two projection operators renaming cn make sure that the cn produced by the translated expressions is the one from π and not the one from e.

4 The input sequence may already be in document order, for example because it resulted from a location path that returned a sorted result[15]. We defer determination of interesting orders in XPath and the resulting optimization of sort operations, as we are primarily concerned with a complete translation, and not yet an optimized one.

3.6. Function Calls

We distinguish between simple function calls, node-set-based function calls and node-set-valued function calls. Simple function calls are characterized by the fact that they neither get node-sets as parameters nor return node-sets, while node-set-based function calls have node-sets as parameters and return simple values. Node-set-valued function calls may return node-sets.

3.6.1. Simple Functions Examples for simple functions in XPath are functions to deal with strings, numbers, or Boolean values (e.g. `string-length`, `floor`, `ceiling`, `true`, `false`, etc.). They are mostly used as subscripts of algebraic operators. Translating simple functions is quite straightforward (f is translated into its algebra counterpart):

$$\mathcal{T}[f(e_1,\dots,e_n)] := f(\mathcal{T}[e_1],\dots,\mathcal{T}[e_n])$$

3.6.2. Node-set-based functions We classify the node-set-based function calls further into aggregate functions and comparison operators between two node-sets. If f is an aggregate function (XPath's sum or count), then we translate it into the corresponding aggregate operator $\mathfrak{A}_f$ of our algebra. $\mathfrak{A}_f$ aggregates the tuples of the input node-set applying the function f and returns a tuple containing the answer. Let e be a node-set value, then[5]

$$\mathcal{T}[f(e)] := \mathfrak{A}_{a;f}(\mathcal{T}[e])$$

For the comparison operators on node-sets it is important to know that they have an existential semantics. That is, if we can find two elements (one in each node-set) that satisfy the condition, the comparison operator returns true. To implement this, we have additional internal aggregation functions $exists()$, $max()$ and $min()$. $exists()$ is boolean-valued and returns false for empty sequences and true otherwise. $max_a()$ and $min_a()$ return the maximum or minimum of an attribute a in a tuple sequence, where for node attributes, each node content is converted to a number by means of the $number()$ function.

For (in)equality, we have

$$\begin{aligned}\mathcal{T}[e_1 = e_2] &:= \mathfrak{A}_{x;exists}(\mathcal{T}[e_1] \ltimes_{cn=cn'} \Pi_{cn':cn}(\mathcal{T}[e_2]))\\ \mathcal{T}[e_1 \neq e_2] &:= \mathfrak{A}_{x;exists}(\mathcal{T}[e_1] \triangleright_{cn=cn'} \Pi_{cn':cn}(\mathcal{T}[e_2]))\end{aligned}$$

For $\theta \in \{<, \leq\}$ (recall that the nodes produced by sequence-valued e_2 are assigned to attribute cn):

$$\mathcal{T}[e_1 \theta e_2] := \mathfrak{A}_{x;exists}(\sigma_{cn\theta\mathfrak{A}_{max_{cn}(\mathcal{T}[e_2])}}\mathcal{T}[e_1])$$

5 Our aggregation operator $\mathfrak{A}$ formally has sequence-valued input *and* output. However, here we use it according to our conventions (Sec. 2.2.2) as an atomic value. We explain in Sec. 5.2 how this conversion is actually implemented.

Finally, for $\theta \in \{>, \geq\}$:

$$\mathcal{T}[e_1 \theta e_2] := \mathfrak{A}_{x;exists}(\sigma_{cn\theta\mathfrak{A}_{min_{cn}(\mathcal{T}[e_2])}}\mathcal{T}[e_1])$$

3.6.3. Node-set-valued functions The only node-set-valued function in XPath 1.0 is $id()$. We translate $id()$ by first converting the input into a sequence of IDs. Then, the individual IDs are dereferenced using a dereference function which converts a single ID string into a node[6]. The result is a sequence-valued expression with the result nodes assigned to cn.

The input conversion depends on whether the input is of type node set or not.

For a node set input e, we just convert the nodes to strings:

$$\mathcal{T}[id(e)] := \chi_{cn:deref(string(c'))}(\Pi_{c':cn}(\mathcal{T}[e]))$$

In the case of an e which is not of type node-set, we convert e to a string and use unnest map with a tokenizing function to return the sequence of embedded string tokens:

$$\mathcal{T}[id(e)] := \chi_{cn:deref(t)}(\Upsilon_{t:tokenize(string(\mathcal{T}[e]))}(\Box))$$

3.7. Constants and Variables

Constants and variables are very easy to translate into our algebra. For the translation of a constant c we have $\mathcal{T}[c] = c$. That means, the expression is left as is and no algebraic operator is necessary to process it. The same applies to XPath $ variables, as they are bound to values before evaluating expressions.

4. Improved Translation

After presenting the canonical translation into the algebra, we go into some details on how to improve the translation step. In particular, Gottlob et al. have shown how XPath expressions can be evaluated in polynomial time in the worst case [9]. In this section we reveal how this can be done in an algebra-based approach.

4.1. Pushing Duplicate Elimination

We divide up all location steps into two different groups: one that potentially produces duplicates (ppd) and one that does not ($\neg ppd$). Axes that belong to ppd are: `following`, `following-sibling`, `preceding`, `preceding-sibling`, `parent`, `ancestor`, `ancestor-or-self`, `descendant`, and `descendant-or-self`. Instead of such a simple axis-wise treatment, we could incorporate the work by Hidders et al. for determining if a *sequence* of location steps will produce duplicates or not [15]. We skip this due to space constraints, because it does not affect asymptotical complexity, and is straightforward to implement.

The canonical translation eliminates duplicates only in a final step to preserve the duplicate-free semantics of XPath location paths. However, duplicates may be generated after every single step. The single final duplicate elimination means that the effect of producing duplicates in several intermediate steps will multiply, as we generate duplicates of duplicates.

Hence, we introduce additional duplicate eliminations after ppd axes. This reduces the input size of the following steps. Also, the duplicate elimination works on smaller data sets. For the translation of a location path $\pi_0/a :: t$ with a step s comprised of axis a and nodetest t this means:

$$\mathcal{T}[\pi_0/a :: t] := \begin{cases} \Pi^D(\mathcal{T}[\pi_0] < \Upsilon_{c:c_0/a::t}(\Box) >) & \text{if } ppd(s) \\ \mathcal{T}[\pi_0] < \Upsilon_{c:c_0/a::t}(\Box) > & \text{else} \end{cases}$$

4.2. Location Paths

When improving on the translation of location paths we have to distinguish between outer and inner paths. An *inner path* appears within a predicate, an *outer path* does not. We discern between these two cases, because we can translate outer paths in a more efficient way deviating from the canonical d-join translation. For inner location paths we run the risk of having to evaluate expressions multiple times for the same context node. We avoid this by memoizing already evaluated paths.

4.2.1. Outer Paths For outer location paths we concatenate the evaluation of the steps, avoiding the overhead of a d-join operator. We replace the singleton scan in the dependent branch with the left subexpression of the d-join. For an outer path π/s we get the following *stacked translation*:

$$\mathcal{T}[\pi/s] := \begin{cases} \Pi^D(\mathcal{T}[s](\mathcal{T}[\pi])) & \text{if } ppd(s) \\ \mathcal{T}[s](\mathcal{T}[\pi]) & \text{else} \end{cases}$$

The linear structure of the stacked translation is shown in Fig. 3, for the path $/a_1 :: t_1/a_2 :: t_2/a_3 :: t_3$.

4.2.2. Inner Paths When looking at inner location paths, we have to distinguish between relative and absolute inner paths. A *relative inner path* gets its context from the corresponding step of the outer path, an *absolute inner path* sets its own context. The translation of the actual inner path then takes place during the translation of the predicate (see also Section 3.3.2).

6 We do not elaborate on the implementation of $deref()$ here, as it depends too much on the details of the storage environment.

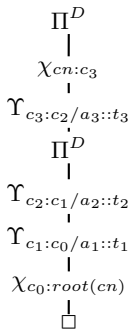

Figure 3. improved stacked translation for $/a_1 :: t_1/a_2 :: t_2/a_3 :: t_3$ with $ppd(a_2 :: t_2)$

Absolute inner paths can be translated like outer paths, while we have to avoid unnecessary work in relative inner paths as follows.

In the XPath expression

$$\pi[\text{count}(./\text{descendant} :: c/\text{following} :: *) = 1000]$$

when evaluating the predicate for the context nodes produced by π, we may reach the same c elements over and over again, computing and counting the nodes produced by the `following` axis multiple times.

For the location path $\pi_0[\pi_1/s]$ the inner path is translated as $\mathcal{T}[\pi_1] < \mathcal{T}[s] >$. We want to avoid computing the right-hand side of the d-join when getting handed a context node from the left hand side for which s was already evaluated before.

In order to avoid this, we apply a memoization strategy using a *MemoX operator* ($\mathfrak{M}$). In contrast to the memoizing function call operator from [13], the MemoX operator is a *sequence-valued* unary operator typically used in the dependent subexpression of a d-join. It is subscripted with a set of variables which are free in its producer expression.

Every time the MemoX operator is evaluated, it checks if the variables have already been bound with these specific values in a prior evaluation. If not, the MemoX operator evaluates the subexpression, and stores the result in an associative data structure with the given variable values as key. Finally it also returns the result to its consumer. If the same variable values have already been used in an earlier evaluation of the MemoX operator, it just looks up the previously computed result and returns it without engaging the producer operator.

In the case of the translation of inner paths, the producer operator is the next location step and the free variable is the current context node from the previous location step. So the translation of the inner path s/π_1 actually looks like:

$$\mathcal{T}[s/\pi_1] :=$$

$$\begin{cases} \mathcal{T}[s] < \mathcal{T}[\pi_1] > & \text{if } \neg ppd(s) \text{ and } \neg ppd(\pi_1) \\ \Pi^D(\mathcal{T}[s] < \mathfrak{M}_{cn}(\mathcal{T}[\pi_1]) >) & \text{if } ppd(s) \text{ and } \neg ppd(\pi_1) \\ \Pi^D(\mathcal{T}[s] < \mathcal{T}[\pi_1] >) & \text{if } \neg ppd(s) \text{ and } ppd(\pi_1) \\ \Pi^D(\mathcal{T}[s] < \mathfrak{M}_{cn}(\mathcal{T}[\pi_1]) >) & \text{if } ppd(s) \text{ and } ppd(\pi_1) \\ \mathcal{T}[s] & \text{if } \pi_1 \text{ is empty} \end{cases}$$

4.3. Predicate Evaluation

4.3.1. Predicates and Stacked Translation In the canonical translation all tuples produced on the right hand side of the d-join for a given tuple on the left hand side belong to the same context. So all contexts are clearly separated from each other by separate evaluations of dependent d-join subexpressions. This makes it easy to determine context position and context size by counting the tuples in one complete evaluation of a dependent subexpression.

In the stacked translation, all tuples belonging to a location step are part of the same tuple stream flowing through the pipeline of operators. The different contexts are separated by the input context nodes; a new context begins whenever the input context node c_{i-1} changes. This requires a slightly different handling of predicate evaluation.

Whenever a c_{i-1} value different from the last processed tuple is detected, the map performing the position count must reset its counter.

We also have to be careful when evaluating predicates of outer locations paths containing `last`. As we have already mentioned in Section 4.2.1, outer location paths are evaluated in a stack-based fashion. The Tmp^{cs} operator now has to be able to recognize the boundaries of the contexts. For this task we define a $\textit{Tmp}_c^{cs}$ operator parameterized with the context node attribute c:

$$\text{Tmp}_c^{cs}(e) := e\Gamma_{cs;c=c';count}\Pi_{c':c}(e)$$

This operator performs the same task as Tmp^{cs}, but does not aggregate the whole input sequence, but only those tuples that were generated for the same context node c.

When evaluating predicates, the new operator is used in the same way as the Tmp^{cs} operator in Sec. 3.3, but is parameterized with the input context node as $Tmp^{cs}_{c_{i-1}}$.

Fig. 4 shows the graphical representation of a $\mathcal{T}[]$ result for a more complex XPath expression.

4.3.2. Avoiding Evaluation of Expensive Predicates We classify the clauses l_{kj} of a predicate p_k into the sets

$$\begin{aligned} \text{cheap}(p_k) &:= \{l_{kj} \mid l_{kj} \text{ is cheap to evaluate}\} \\ \text{exp}(p_k) &:= \{l_{kj} \mid l_{kj} \text{ is expensive to evaluate}\} \\ \text{pos}(p_k) &:= \{l_{kj} \mid l_{kj} \text{ contains } \texttt{position()}, \\ &\qquad \text{but no } \texttt{last()}\} \\ \text{last}(p_k) &:= \{l_{kj} \mid l_{kj} \text{ contains } \texttt{last()}\} \end{aligned}$$

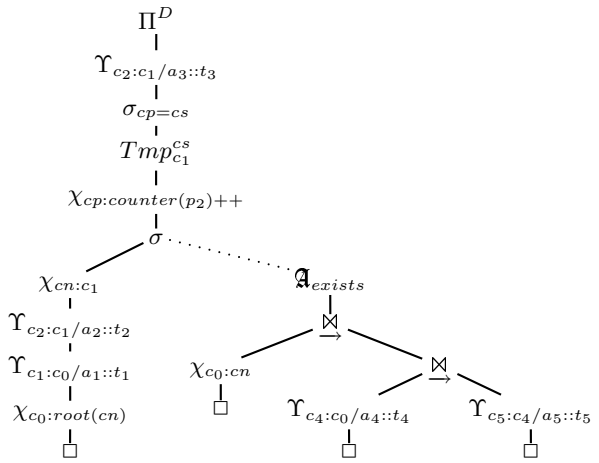

Figure 4. $/a_1 :: t_1/a_2 :: t_2[a_4 :: t_4/a_5 :: t_5][position() = last()]/a_3 :: t_3$

For classification into cheap() and exp(), a simple cost model is used which considers the number of instructions that are necessary to evaluate a clause. For the translation of a predicate $p_k = l_{k1} \ldots l_{km_k}$, this means

$$\Phi[p_k] := \sigma^{mat}_{exp(p_k)} \circ \overbrace{\sigma_{cheap(p_k) \cap last(p_k)} \circ \mathrm{Tmp}^{cs}}^{\text{only for } \texttt{last}} \circ \sigma_{cheap(p_k) \setminus last(p_k)} \circ \underbrace{\chi_{cp:counter(p_k)++}}_{\text{only for } \texttt{pos} \text{ or } \texttt{last}}$$

Each expensive expression e in a clause l_{kj} that is a member of $\texttt{exp}(p_k)$ is replaced by a variable v. We compute the value of v with the help of χ_{mat} operators, which memoizes function evaluation results similar to the approach by Hellerstein and Naughton [13]. In the translation above, $\sigma^{mat}_{exp(p_k)}$ is an abbreviation for this sequence of χ_{map} operators and the final selection.

In the above translation, Tmp^{cs} has to be replaced by Tmp^{cs}_c if the predicate occurs in a stacked translation.

5. Implementation

5.1. Compiler

The translation was implemented as an XPath compiler module written in C++, taking XPath expressions as strings and generating an execution plan for the NQE (see below). The compilation process comprises six steps, listed here with some of the tasks they perform: (1) Parsing (generating an abstract syntax tree (AST)) (2) Normalization (classifies and sorts predicates as explained in Sec. 3.3 and 4.3) (3) Semantic analysis (4) Rewrite (constant folding) (5) Translation into algebra (6) Code generation (generate NQE execution plan). The query is handed from step to step using a single data structure, starting out as an AST which is annotated and modified until it has become an algebraic expression. The resulting expression is traversed by step (6), which returns an execution plan in the NQE syntax. A detail worth noting is that our translation includes a lot of map and projection operations, particularly to guarantee that the context node attribute is always called cn. The compiler does not emit actual copy operations in these cases. Instead, an attribute manager which is part of the compiler ensures that code emitted for aliased attribute accesses uses the proper memory locations directly.

5.2. Physical Algebra

The Natix system's Query Execution Engine (NQE) implements the logical algebra from Sec. 2.2 in C++. Below, we focus on implementation aspects relevant for XPath. More details can be found in [6] and [14].

5.2.1. Iterators All the sequence-valued operators in our logical algebra (Fig. 1) have a corresponding implementation as an *iterator* [10] in the physical algebra. Whenever possible, they avoid to copy and/or materialize intermediate results, passing them by reference and/or in a pipelining fashion.

5.2.2. Natix Virtual Machine The remaining (i.e. non-sequence valued) operators of our logical algebra are implemented using assembler-like programs interpreted by the Natix Virtual Machine (NVM). XPath basic type functions and operators are evaluated using single NVM commands or small command sequences.

Location step navigation and node tests are performed via NVM commands that directly access the persistent representation of the documents in the Natix page buffer, thus avoiding an expensive representation change into a separate main memory format. In the buffer, the XML documents are stored in recoverable, updatable form which does not require a fixed DTD. There are also NVM commands for access to text node contents. However, we transcode the stored, space-saving string encoding to UTF-16, which is the encoding used for strings in NVM.

5.2.3. Nested Iterators NVM programs are primarily used to evaluate non-sequence-valued subscripts of iterators, and the NVM commands operate on tuples. Sometimes subscripts also need to evaluate XPath functions that have sequences as input, and to convert the sequence-valued result of the $\mathfrak{A}$ operator into an atomic value.

The NVM provides commands that can access results of nested iterators.

5.2.4. Context Size Operators In Sec. 3.3 and Sec. 4.3, we introduced special operators Tmp^{cs} and Tmp^{cs}_c which determine the context size and concatenate it to all output tuples. The logical definition of these operators requires the input sequence twice, once for determining the number of tuples, and once to return the actual result annotated by the size.

The actual implementation does not evaluate the input context twice. Instead, each context is evaluated once and materialized. Note that the input tuples for Tmp^{cs}_c always contain cp, the position counter. The cp value of the final tuple equals the context size cs, which is remembered. When delivering the result, the materialized sequence is reread, adding cs to each tuple as it is returned.

Tmp^{cs} and Tmp^{cs}_c only differ in how they determine the input context to materialize. Tmp^{cs} counts the complete input sequence, while Tmp^{cs}_c only materializes those input tuples generated for the same input context node. The materialization stops when the input context node attribute c changes (compare Sec. 4.3).

Actually, there is just one implementation Tmp^{cs}_c which covers Tmp^{cs} as a special case.

5.2.5. Smart Aggregation The aggregation functions used by $\mathfrak{A}$ are also implemented as small NVM programs. The interface between the $\mathfrak{A}$ operator and these programs allows to signal a premature end of the aggregation. For example, when evaluating an $exists()$ function, it is not necessary to evaluate the complete argument sequence. If one tuple is found, the remaining input sequence may be ignored, and the $\mathfrak{A}$ operator may return $true$.

6. Evaluation

Our ultimate goal in providing a complete algebraic translation is performance. While we do not discuss advanced optimization techniques on the algebraic level in this paper, another performance-related aspect of the algebraic approach is the fact that queries can be executed in a scalable way through an iterator-based approach.

To verify that this goal has been met, we compared our implementation against some purely main-memory based XPath interpreters. We chose those freely available interpreters which support the complete XPath specification, including all axes, namely `xsltproc` [19] and Xalan [22].

We are aware that the following is not a comprehensive performance evaluation, and leaves open a lot of questions. The measurements below are intended to give a proof-of-concept of our approach, and we gathered them only to make sure that we are on the right track.

no.	path
1	`/child::xdoc/desc::*/anc::*/desc::*/@id`
2	`/child::xdoc/desc::*/pre-sib::*/fol::*/@id`
3	`/child::xdoc/desc::*/anc::*/anc::*/@id`
4	`/child::xdoc/child::*/par::*/desc::*/@id`

Figure 5. Queries against generated documents

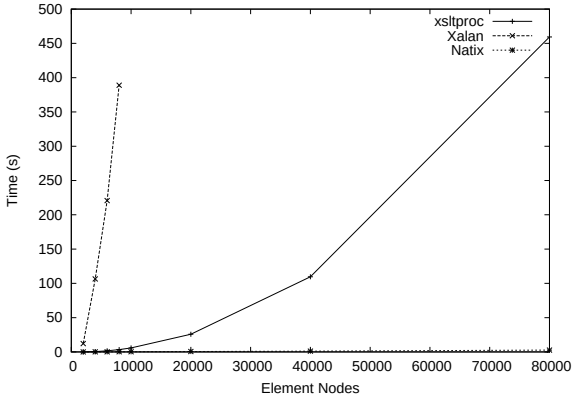

Figure 6. results for path number 1

6.1. Environment

The environment used to perform the experiments consisted of a PC with an Intel Pentium 4 CPU at 2.80GHz and 1 GB of RAM, running Linux 2.6.4. The Natix C++ library and the test executable were compiled with gcc 3.3 at the O2 optimization level.

6.2. Results

Below, we list the time needed to compile and execute a query. To make them comparable across the different evaluators, the times do not include the time to parse/load the document. The measurements are averaged over several runs.

6.2.1. Generated Documents The documents on which the queries in Fig. 5 are executed were generated. They differ in the number of elements, fanout and document depth. The document generator follows a breadth first algorithm and fills every depth of the document with the given fanout until the maximum number of elements or depth is reached. The root element of every document has the name `xdoc`. Every element contains an attribute `id` which is consecutively numbered.

The concrete documents between 2000-8000 elements were generated with a fanout of six and a depth of four. The documents between 10000-80000 were generated with a fanout of ten and a depth of five.

The queries were obtained by systematically generating all XPath location paths of length 3 with a node test check-

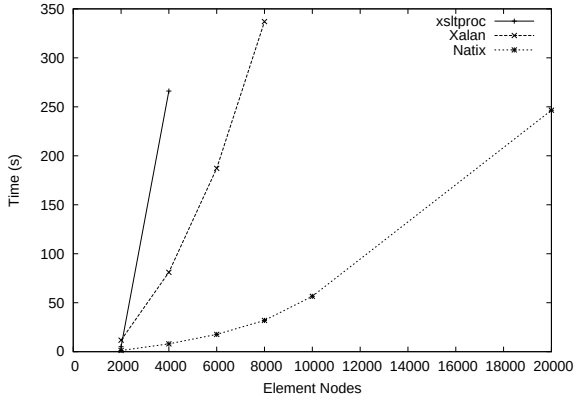

Figure 7. results for path number 2

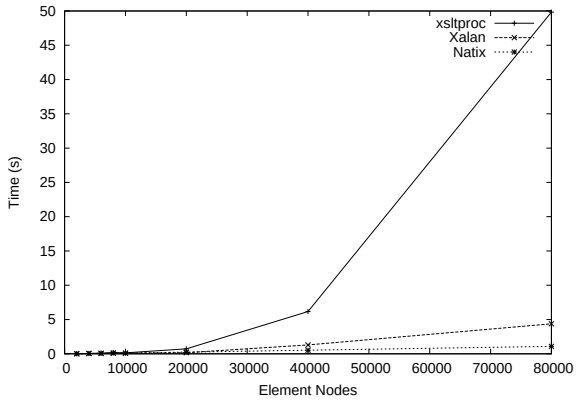

Figure 8. results for path number 3

ing for any element node in each step. There are several typical patterns in the results, and we selected sample queries as examples for these patterns.

Fig. 6–Fig. 9 show the selected results. Without going into detail, we observe (1) that we can keep up with the performance of the main-memory based interpreters, (2) the high memory requirements sometimes cause the main-memory interpreters to fail on large documents (this is the reason the curves sometimes stop before reaching the end of the x-axis), and (3) the constants in the asymptotic behavior of the algebraic approach are promisingly small.

In some queries like the one in Fig. 9, one or both main-memory evaluators outperform Natix by a constant factor. Profiling NQE has provided us with hints on how to lower this constant factor.

6.2.2. DBLP data Fig. 10 shows execution times for queries executed on DBLP data collected in one big XML document [18]. This document has a size of 216mb.

`xsltproc` was not able to load the document, probably due to memory requirements.

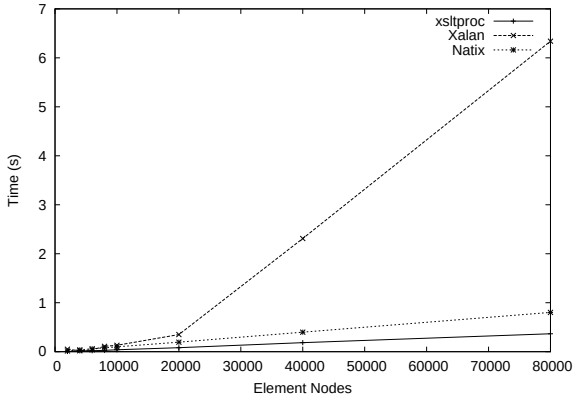

Figure 9. results for path number 4

As above, the results are promising, as we can compete with a main-memory based evaluation. For some queries (those below the horizontal line in the center) we are slower. Again, profiling points to engineering details in NQE as a reason, which we plan to correct in the future.

	time[s]	
path	Xalan	Natix
/dblp/article/title	6.50	3.97
/dblp/*/title	17.73	8.10
/dblp/article[position() = 3]/title	24.51	1.51
/dblp/article[position() $<$ 100]/title	25.22	1.55
/dblp/article[position() = last()]/title	23.99	2.22
/dblp/article[position()=last()-10]/title	24.32	2.31
/dblp/article/title | /dblp/inproceedings/title	157.98	14.23
/dblp/article[count(author)=4]/@key	0.9	2.91
/dblp/article[year='1991']/@key | /dblp/inproceedings[year='1991']/@key	3.90	8.69
/dblp/*[author='Guido Moerkotte']/@key	4.2	9.78
/dblp/inproceedings [@key='conf/er/LockemannM91']/title	3.22	4.28
/dblp/inproceedings [author='Guido Moerkotte'] [position()=last()]/title	4.59	6.71

Figure 10. Queries against DBLP

7. Conclusion and Future Work

In this paper, we explained how to translate XPath queries into algebraic expressions.

The proposed translation method covers the complete set of XPath features including all axes, position-based predicates, nested paths, filter expressions, general path expressions and node-set functions.

Apart from providing an effective translation as a first step, we were also concerned with efficiency. In a second step, we extended our simple approach, achieving polynomial worst-case complexity. To this end, we incorporated the memoization techniques pioneered by Gottlob et al. [8] in the context of XPath interpreters.

We are not aware of any other algebraic approach to XPath evaluation covering all of these aspects.

We have not yet unleashed the full power of a cost-based algebraic optimizer. Before doing so, we wanted to verify that the nitty-gritty details that tend to come up in practice can be solved, and that our approach can hold its ground performance-wise.

Hence, we implemented an XPath compiler based on the concepts outlined in this paper. A complementary iterator-based physical algebra was used to evaluate the generated algebraic query plans. First measurements demonstrate that our approach is viable.

Having established that an algebraic approach to XPath is reasonable, we can now turn to the next challenges. While there already are papers about XPath-specific algebraic rewriting techniques [14, 20], much remains to be done. Areas that come to mind are cost functions, schema-based rewritings [2, 17], equivalences, using properties of the intermediate results to avoid duplicate elimination and sorting [15], and an increase of the search space, e.g. by using magic sets and indexes. In addition, the upcoming XPath 2.0 semantics are different from 1.0, and our compiler and algebra need support for more complex types. Further in the future, XQuery with its complex result construction and nesting capabilities is also a target of the authors' algebraic ambitions.

Acknowledgments: We thank the anonymous referees for their helpful hints on improving this paper.

References

[1] Catriel Beeri and Yariv Tzaban. SAL: An algebra for semistructured data and xml. In *WebDB (Informal Proceedings)*, pages 37–42, 1999.

[2] Klemens Bohm, Karl Aberer, M. Tamer Ozsu, and Kathrin Gayer. Query optimization for structured documents based on knowledge on the document type definition. In *Advances in Digital Libraries*, pages 196–205, 1998.

[3] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal XML pattern matching. In *ACM SIGMOD*, pages 310–321, 2002.

[4] James Clark and Steve DeRose. XML path language (XPath) version 1.0. Technical report, World Wide Web Consortium (W3C) Recommendation, 1999.

[5] Mary F. Fernandez, Jérôme Siméon, Byron Choi, Amélie Marian, and Gargi Sur. Implementing Xquery 1.0: The Galax experience. In *VLDB Conf.*, 2003.

[6] Thorsten Fiebig, Sven Helmer, Carl-Christian Kanne, Guido Moerkotte, Julia Neumann, Robert Schiele, and Till Westmann. Anatomy of a Native XML base management system. *VLDB Journal*, 11(4):292–314, 2003.

[7] Daniela Florescu and Donald Kossmann. Storing and querying xml data using an rdmbs. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[8] G. Gottlob, C. Koch, and R. Pichler. Xpath query evaluation: Improving time and space efficiency. In *ICDE*, pages 379–390, 2003.

[9] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing XPath queries. In *VLDB Conf.*, pages 95–106, 2002.

[10] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[11] Torsten Grust. Accelerating XPath location steps. In *SIGMOD Conference*, pages 109–120, 2002.

[12] Torsten Grust, Sherif Sakr, and Jens Teubner. Xquery on sql hosts. In *VLDB Conference*, pages 252–263, 2004.

[13] Joseph M. Hellerstein and Jeffrey F. Naughton. Query execution techniques for caching expensive methods. In *ACM SIGMOD*, 1996.

[14] S. Helmer, C.-C. Kanne, and G. Moerkotte. Optimized translation of XPath expressions into algebraic expressions parameterized by programs containing navigational primitives. In *WISE*, pages 215–224, 2002.

[15] Jan Hidders and Philippe Michiels. Avoiding unnecessary ordering operations in xpath. In *DBLP*, pages 54–74, 2003.

[16] H. Jagadish, L. Lakshmanan, D. Srivastava, and K. Thompson. Tax: A tree algebra for xml. In *Proceedings of DBPL'01*, 2001.

[17] April Kwong and Michael Gertz. Schema-based optimization of xpath expressions. Technical report, University of California Davis, 2002.

[18] Michael Ley. Dblp xml records. http://dblp.uni-trier.de/xml/.

[19] libxslt 1.1.2—the xslt c library for gnome. http://xmlsoft.org/XSLT/.

[20] Dan Olteanu, Holger Meuss, Tim Furche, and Francois Bry. XPath: Looking forward. In *EDBT Workshop on XML Data Management*, 2002.

[21] Stelios Paparizos, Shurug Al-Khalifa, Adriane Chapman, H. V. Jagadish, Laks V. S. Lakshmanan, Andrew Nierman, Jignesh M. Patel, Divesh Srivastava, Nuwee Wiwatwattana, Yuqing Wu, and Cong Yu. TIMBER: a native system for querying XML. In *ACM SIGMOD*, 2003.

[22] Apache XML Project. Xalan C++ version 1.6. http://xml.apache.org/xalan-c/index.html, 2003.

[23] Carlo Sartiani and Antonio Albano. Yet another query algebra for xml data. In *IDEAS*, pages 106–115. IEEE Computer Society, 2002.

[24] Shurug Al-Khalifa Shurug, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural joins: A primitive for efficient xml query pattern matching. In *ICDE*, pages 141–152, 2003.