Informatik - Forschung und Entwicklung manuscript No.
(will be inserted by the editor)

Christian Mathis

Extending a Tuple-Based XPath Algebra to Enhance Evaluation Flexibility

Eingegangen: date / Angenommen: date

Zusammenfassung In der Literatur der vergangenen Jahre wurde der algebraischen Behandlung der XML-Anfragesprache XPath wenig Bedeutung zugemessen. Eine löbliche Ausnahme bildet die Natix-Algebra (NAL), welche auf präzise Weise die Übersetzung einer XPath-Anfrage in einen algebraischen Ausdruck definiert, und somit die Tür zur algebraischen Optimierung dieser Anfragesprache öffnet. Bei genauerer Betrachtung verpasst es NAL jedoch, bekannte und vielversprechende Auswertungsalgorithmen, wie zum Beispiel den „Holistic Twig Join“, in den Übersetzungsprozess einzubeziehen. Die in diesem Artikel vorgeschlagene Einführung eines logischen strukturellen Verbundes („Structural Join“) behebt diese Schwachstelle und erlaubt es, einen NAL-Ausdruck in eine physische Algebra zu übersetzen, die genau diese fehlenden Auswertungsalgorithmen enthält. Zusätzlich werden wichtige Regeln zur Entschachtelung von XPath-Anfragen eingeführt. Mit Hilfe des „XML Transaction Coodinators“ (XTC) – unserem Protoyp eines nativen XML-Datenbanksystems – werden die zu erwartenden Effizienzsteigerungen nachgewiesen.

Schlüsselwörter XML-Anfrageverarbeitung · XPath-Algebra · XPath-Entschachtelung

Abstract Over the recent years, very little effort has been made to give XPath a proper algebraic treatment. One laudable exception is the Natix Algebra (NAL) which defines the translation of XPath queries into algebraic expressions in a concise way, thereby enabling algebraic optimizations. However, NAL does not capture various promising core XML query evaluation algorithms like, for example, the Holistic Twig Join. By integrating a logical structural join operator, we enable NAL to be compiled into a physical algebra containing exactly those missing physical operators. We will provide several important query unnesting rules and demonstrate the effectivity of our approach by an implementation in the XML Transaction Coordinator (XTC)—our prototype of a native XML database system.

Keywords XML Query Processing · XPath Algebra · XPath Unnesting

CR Subject Classification H.2.4 · D.3.4

This work has been supported by the Rheinland-Pfalz cluster of excellence “Dependable Adaptive Systems and Mathematical Modeling” (see http://www.dasmod.de). The article at hand is an extended version of [20].

Christian Mathis
AG Datenbanken und Informationssysteme
Universität Kaiserslautern
P. O. Box 3049
67653 Kaiserslautern
Tel.: +49-631-2053282
Fax: +49-631-2053299
E-Mail: mathis@informatik.uni-kl.de

1 Introduction

There is one core task, common to almost all XML query languages: the matching of path patterns against XML documents. The performance of an XML query language processor intrinsically depends on its path evaluation engine, because path matching is a frequent and expensive operation. Path matching occurs frequently, because even multiple paths are often defined in a single query. And it is expensive, because path evaluation requires physical access to the document, in contrast to almost all other constructs of an XML query language, which are evaluated on the output generated by path matchings. In spite of the many algebra proposals regarding the standard XML query language *XQuery* [16, 21, 25], its path-related sublanguage XPath has unfortunately not gained as much attention. However, because of the above reasons, we believe that XPath should be furnished with an algebraic basement, too: It is the core XML data access mechanism in XQuery (and also XSLT) and it is itself a complex language to evaluate leaving a lot of space for algebraic optimizations. In this paper, we will extend the Natix Algebra (NAL) [4], which is one of the few algebras specifically dealing with the compilation of XPath.

What is missing in NAL? We observed that somewhat in parallel to the progress being made in the XML algebra community, a plethora of core algorithms for XML query evaluation as well as indexing techniques have been published that qualify as *physical*[1] XML query operators. Among them, the most prominent representatives are the Structural Join [1, 8, 18, 19], the Holistic Twig Join [6, 11], as well as further operators to match general twigs [12], and the various path indexes like, for example, the structural D(k) index [7] or the hybrid content and structure (CAS) index [15]. While being introduced in the context of *tree-based* algebras [16, 17], very little attempt has been made to integrate these concepts into a *tuple-based* XML algebra, such as NAL [21]. You may think, why bother, the combination of a tree-based algebra with the holistic twig join works perfectly, so where is the need for a further XML algebra? We believe that the data model of tuple algebras is more general than the one of tree algebras and, therefore, certain XML query language constructs can be handled more suitably. For example, we do not know how a non-tree intermediate result, like pairs of siblings, is represented without introducing an artificial parent node (which has to be handled by subsequent operators). Furthermore, all major RDBMS vendors are currently integrating XML query capabilities into their (tuple-based) relational query engines. For them, the integration of an equally tuple-based XPath/XQuery algebra would be a natural thing to do[2]. That is why we favor tuple algebras and think the integration of the above mentioned physical operators is of great importance.

In this article, we will elaborate on the algebraic treatment of XPath. We will introduce a *logical* structural join operator into NAL and provide essential rewriting rules to convert an algebraic expression into a format facilitating the mapping onto the existing physical XML strutural join and holistic twig join operators. The extended algebra will be named NAL$^{\text{STJ}}$ (STJ for "structural join").

1.1 XML Algebras in the Literature

To give an overview over all XML algebra proposals is certainly out of the scope of this paper. Therefore we can only focus on a few algebras that explicitely tackle the issue of path matching and algebraic XPath treatment.

The TAX and TLC algebras [16, 17] evolve from an analogy between relations and trees. In the relational algebra, each operator consumes and produces sets of tuples (relations), whereas sequences of XML data trees are the basic unit of processing in TAX/TLC, i. e., TAX/TLC is a tree-based algebra. A core concept to all operators are pattern trees. They can be used, for example, to define a query tree structure for a selection operator that matches the pattern tree against a document, thereby producing a sequence of so-called witness trees. Each witness tree in the result sequence corresponds to a match. The above mentioned physical algorithms Structural Join and Holistic Twig Join, are core algorithms in the TAX/TLC physical algebra, because they do the job of pattern tree matching. TAX/TLC provides a "natural" way to process XML trees, because it is based on XML trees as intermediate results. However, its expressive power is definitely too limited for the evaluation of XPath queries: only the descendant and child axis are supported for the definition of a pattern tree.

The Natix Algebra (NAL) [21] takes a different approach, because it abstracts from trees as intermediate result structures. NAL operates on sequences of (homogeneous) tuples, each tuple consisting of a set of attribute-to-value mappings. Similar to the notion of the *evaluation context* defined in the W3C Formal Semantics [9], these mappings keep track of the dynamic variable bindings during query processing. Reference [4] describes the translation of an arbitrary XPath expression into NAL. Because our article heavily relies on NAL, we will sufficiently introduce the algebra and its capabilities in the following.

The algebra presented in [25]—called RSF[3] algebra in the following—employs a hybrid approach. Its expressions contain both operator types: Tree-based operators are introduced for intermediate XML tree handling and tuple-based operators to control the flow of tuple streams generated by XQuery's *for* and *let* expressions. To ensure the compatibility between these two types, special conversion operators (`MapToItem` and `MapFromItem`[4]) have to be embedded into an algebra expression. This technique avoids tuple flattening which is often required in NAL. Because RSF expressions are generated from the XQuery core representation defined in [9], the whole extent of XPath is covered. In parallel to [20], the RSF group has published an approach to algebraically "find" twig-like search patterns (branching path patterns) in general XQuery expressions [14, 23]. Those patterns are mapped onto a logical twig join operator. While their work and ours aims at the same target, the algebraic equivalences in this article have not been published in RSF so far. Their integration would be possible, though, because RSF and NAL are closely related.

1.2 A Brief Example in NAL

In this section, we will give a brief example in the Natix algebra and point out its strengths and weaknesses. Let us consider the expression $/a_1 :: t_1/a_2 :: t_2\ [position() =$

[1] By "physical" we mean that these operators could be part of a physical XML algebra.

[2] See also [3] for academic research activities in this area.

[3] Named after the last names of the paper's authors.

[4] `MapToItem` converts a sequence of tuples to a sequence of XML trees, while `MapFromItem` works in the opposite direction.

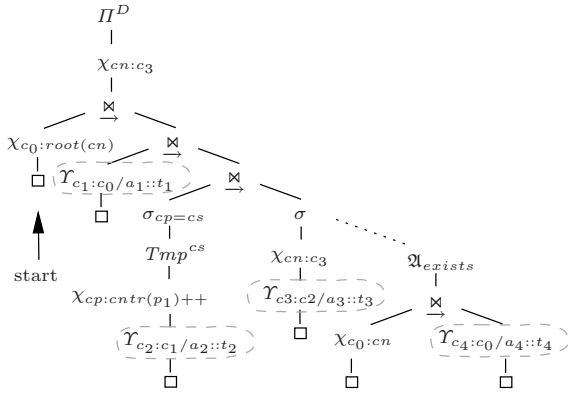


Fig. 1 NAL Example

$last()]/a_3 :: t_3$ $[a_4 :: t_4]$ depicted in Figure 1. The evaluation starts with the singleton scan operator ($\Box$) which creates a singleton sequence containing an empty tuple. It triggers the map operator (χ) to bind the root node of the queried tree to the c_0 attribute of a new tuple. This tuple, in turn, is consumed by the first D-join operator. The D-Join ($\overset{\bowtie}{\rightarrow}$—or $\langle\rangle$ in the textual representation) is similar to XQuery's *for* construct: for each tuple t in the left input sequence, the dependent right expression is evaluated, binding t's attributes to free variables in the right expression (here c_0). Then, the intermediate result calculated for the dependent subexpression is joined with t. In our example, the dependent expression is again a D-Join operator whose left subexpression is an unnest map operator (Υ). This operator is a shortcut for a map operator (χ) followed by an unnest operator (μ). In NAL, Υ is mainly used for the calculation of path axes. Starting from c_0 the path expression $a_1 :: t_1$ is evaluated to a single sequence (using χ) which is immediately unnested (by μ). Together with the D-Join, this results in the above mentioned "flattening".

A predicate is translated into a selection operator (σ), where the predicate's subexpression is compiled into σ's subscript. NAL operators may be arbitrarily nested in this fashion. For each input tuple, the subscript is evaluated. For almost all predicates, certain measures have to be taken to ensure the evaluability of σ's subscript: In case of a relative path expression, the current context variable *cn* has to be provided explicitly. This is accomplished by the two map operators $\chi_{cn:c_3}$ and $\chi_{c_0:cn}$, the first one binding c_3 to the context variable and the second one "transferring" *cn* into the variable c_0 of the local context. For positional predicates, the current context position and the context size have to be calculated. This is the task of the special operators $\chi_{cp:cntr(p_1)++}$ and Tmp^{cs}. The first operator simply counts the tuples in its input and attaches a new attribute *cp*, containing the current position, to them. Tmp^{cs} buffers its input to calculate the total number of tuples in the context, before it attaches attribute *cs*, containing this number, to each tuple. The aggregation operator $\mathfrak{A}$ evaluates aggregate functions, e. g., $min()$, $max()$, etc. More sophisticated predicates, for example existential comparisons, are possible, too. Finally, the resulting context node is produced by a map operator, and duplicate elimination (Π^D) is applied to comply with the XPath semantics.
NAL provides a concise algebraic basement for XPath (1.0 [28]) evaluation. The XPath-to-NAL translation process is described in [4] in great detail. Additionally, the authors provided some optimization techniques like stacked translation for outer paths, duplicate-elimination push down, and memoization[5]. In [5], certain algebraic equivalences were shown, which enable unnesting of queries with semi-correlated XPath predicates[6].

1.3 Problem Statement

In spite of the progress being made in NAL, we believe that there is still room for optimization: Our first observation is that the evaluation of a NAL expression generates almost the same data flow as its equivalent normalized in W3C's XQuery Core Language. As an example, consider the evaluation of the select operator σ in Figure 1: It is evaluated for each context node provided by the unnest map operator $\Upsilon_{c3:c2/a_3::t_3}$. This implies *node-at-a-time* calculation of the path step, embedded in the selection subscript. However, many publications [1, 6, 8] have pointed out that *set-at-a-time* processing of path steps provides better performance in most cases. Another example regarding the generated data flow arises from the order in which the path processing steps are evaluated. Like in XQuery Core, NAL evaluates path steps from left to right. However, as [27] has shown, a reordering of path step evaluations can substantially improve the query processing performance.

As a second point, we observe that the logical-to-physical operator mapping presented in [4] does not take important classes of physical operators into account, like the Structural Join and the Holistic Twig Join[7]. Essentially, these operators provide the above mentioned capability to process path steps in a set-at-a-time manner. There is reasonable doubt that, in the face of complex queries, the algebraic representation can facilitate a mapping onto a physical algebra containing exactly these operators. We draw this doubt from the fact that nested path expressions are "hidden" in subscripts of selection operators. Furthermore, logically related subexpressions, e. g., the compiled parts of the path steps like $a_1 :: t_1$, are "scattered" across the operator tree (shown

[5] These optimizations have not been executed on our example, which is presented in the canonical translation.

[6] Queries with semi-correlated predicates have the form $p = e_1[e_2 \theta e_3]$, where either e_2 or e_3 is a path expression depending on p's outer—or global—context

[7] Although we recognize the hint towards that direction given in [22], we did not find any approach that properly introduces structural joins in NAL.

by the encircled areas in Figure 1). Under the assumption that the above query contains only steps referring to the child and descendant axis, a reasonable evaluation approach—at the physical level—would be the application of a *single* holistic twig join operator, followed by a subsequent selection. However, from the given representation it is unclear, how the mapping onto this holistic twig join operator can be accomplished.

1.4 Our Contribution

Our overall goal is to integrate the above mentionend important classes of physical evaluation operators like Structural Join, Holistic Twig Join, and path index access into NAL's physical algebra. However, as a first step we have to "prepare" NAL at the logical level in a way facilitating this integration. In this article we will

- introduce the structural join operator to the NAL algebra,
- provide rules to convert a NAL expression from its canonical representation into its NAL^{STJ} equivalent containing structural joins,
- develop rewriting rules for predicate unnesting, and
- finally show the impact of our approach on the query processing performance in the XML Transaction Coordinator (XTC)—our prototype of a native XML DBMS.

With the introduction of structural joins, we add another promising evaluation strategy for XML queries to the Natix Algebra: set-at-a-time (or bulk) processing. Due to the many available alternatives for structural join implementation (i. e., stack based [1], hash based [19], index based [8], locking aware [18], etc.), we multiply the number of possible (physical) query plans. This results in a larger search space (hopefully containing a better plan) and in an increased evaluation flexibility. Furthermore, our predicate unnesting rules will facilitate the mapping onto more powerful physical operators like the holistic twig join (which can also evaluate *and*, *or*, and *not* predicates) and path-index lookups, because they expose path processing steps hidden in selection subscripts. Additionally, unnesting enables structural join reordering to prise off the inflexible left-to-right path evaluation. We expect our operator plans to be scalable, though consisting of a large number of joins, because, in contrast to the join implementations in the relational algebra, structural joins are evaluatable in linear time [1].

The question, whether a particular query or predicate should be evaluated node-at-a-time or set-at-a-time, cannot be answered without an (at least rough) estimation of the document's structure. This is clearly not the aim of this article. We just want to provide the necessary algebraic preconditions to enable both evaluation strategies in NAL.

In the following, we will not consider questions arising during plan generation, i. e., during the logical-to-physical operator mapping. Specifically, we will neither show how a holistic twig join can be employed to replace a set of structural join operators, nor how the order of structural joins can be selected [27]. Here, we only want to facilitate the treatment of these important questions by introducing the structural join operator.

The remainder of this article is organized as follows: Sect. 2 provides an overview over the Natix algebra, which we will extend in Sect. 3. The rule-based rewriting of NAL into its extended version is described in Sect. 4, before Sect. 5 introduces the core set of algebraic equivalences for query unnesting. Sect. 6 provides several rules for structural join push down. We conclude this article with a quantitative analysis in Sect. 7.

2 NAL in a Nutshell

For your convenience, we repeat the basic definitions from [4]: NAL operates on sequences of homogeneous sets of attribute-value mappings (tuples) t, each t having the same set of attributes (schema) denoted $A(t)$. Attribute values may be sequences, thus NAL allows arbitrary nesting. The empty sequence is denoted as ϵ or $\langle\rangle$. For tuple modification, NAL provides the primitives $[\cdot]$ (tuple construction), $\circ$ (tuple concatenation[8]), and $|_A$ (attribute projection). The notation $t.a$ describes the access to tuple t's attribute a. $A(e)$ and $F(e)$ denote the schema and the set of free variables of an algebra expression e. Applied to sequences, the functions $e_1 \oplus e_2$, $\alpha(e)$, and $\tau(e)$ return the concatenation ($\oplus$), the first tuple of the sequence (α), and the remainder of the sequence (τ). If e is a sequence of non-tuple values, $e[a] = [a : \alpha(e)] \oplus \tau(e)[a]$ returns a sequence of tuples $[a : e_i]$, where e_i is a tuple of e. An overview over all relevant NAL operators can be found in Table 1. To support the required ordering in XML, all unary operators—except *Sort*—keep the order of their input sequences intact. The binary operators cross product ($\times$) and D-Join ($\langle\rangle$) have nested-loop semantics. The projection operator (Π) has two variants for duplicate elimination (Π^D) and renaming ($\Pi_{a':a}$).

3 Extending NAL to NAL^{STJ}

For our NAL extension NAL^{STJ}, we introduce some new operator definitions and modify a few existing ones. We want to keep NAL^{STJ} backward compatible, i. e., an expression in NAL shall also be an expression in NAL^{STJ}. The new or modified operators are: the structural selection and the structural join, node sequence access, nesting, reverse, group reverse, group sort, context size and context position calculation, and finally sequence-based merge and intersect.

[8] Note, the $\circ$ operator is overloaded and also used for the composition of functions.

Table 1 Relevant NAL Operators taken from [4]

Operator	Definition
Selection	$\sigma_p(e) := \begin{cases} \alpha(e) \oplus \sigma_p(\tau(e)) & : \quad p(\alpha(e)) = true \\ \sigma_p(\tau(e)) & : \quad else \end{cases}$
Projection	$\Pi_A(e) := \alpha(e)\|_A \oplus \Pi_A(\tau(e))$
Map	$\chi_{a:e_2}(e_1) := \alpha(e_1)\|_{Attr(e_1)\setminus\{a\}} \circ [a : e_2(\alpha(e_1))] \oplus \chi_{a:e_2}(\tau(e_1))$
Cross Product	$e_1 \times e_2 := \left((\alpha(e_1))\overline{\times}e_2\right) \oplus (\tau(e_1)\overline{\times}e_2)$
D-Join	$e_1\,\langle e_2\rangle := \alpha(e_1)\overline{\times}e_2(\alpha(e_1)) \oplus \tau(e_1)\,\langle e_2\rangle$
Product	$t_1\overline{\times}e_2 := (t_1 \circ \alpha(e_2)) \oplus (t_1\overline{\times}\tau(e_2))$
Semi-Join	$e_1 \ltimes_p e_2 := \begin{cases} \alpha(e_1) \oplus (\tau(e_1) \ltimes_p e_2) & : \quad \exists x \in e_2 : p(\alpha(e_1) \circ x) \\ \tau(e_1) \ltimes_p e_2 & : \quad else \end{cases}$
Anti-Join	$e_1 \rhd_p e_2 := \begin{cases} \alpha(e_1) \oplus (\tau(e_1) \rhd_p e_2) & : \quad \nexists x \in e_2 : p(\alpha(e_1) \circ x) \\ \tau(e_1) \rhd_p e_2 & : \quad else \end{cases}$
Unnesting	$\mu_g(e) := (\alpha(e)\|_{\overline{\{g\}}} \times \alpha(e).g) \oplus \mu_g(\tau(e))$
Unnest-Map	$\Upsilon_{a:e_2}(e_1) := \mu_g(\chi_{g:e_2[a]}(e_1))$
Binary Grouping	$e_1 \Gamma_{g;A_1\theta A_2;f} e_2 := \alpha(e_1) \circ [g : G(\alpha(e_1))] \oplus (\tau(e_1)\Gamma_{g;A_1\theta A_2;f} e_2)$ where $G(x) := f(\sigma_{x\|_{A_1}\theta A_2}(e_2))$
Unary Grouping	$\Gamma_{g;\theta A;f}(e) := \Pi_{A:A'}(\Pi^D_{A':A}(\Pi_A(e))\Gamma_{g;A'\theta A;f} e$
Aggregation	$\mathfrak{A}_{a:f}(e) := [a : f(e)]$
Sort	$Sort_a(e) := Sort_a(\sigma_{a<\alpha(e).a}(\tau(e))) \oplus \alpha(e) \oplus Sort_a(\sigma_{a\geq\alpha(e).a}(\tau(e)))$
Context Size	$Tmp^{cs}(e) := \mathfrak{A}_{cs;count}(e)\overline{\times}e$
Singleton Scan	$\Box := \{[]\}$

Structural Selection. The structural selection, i. e., the selection of a tuple based on some structural predicate, is embedded by extending the NAL selection operator from Table 1:

$$\sigma_p(s) := \begin{cases} \alpha(s) \oplus \sigma_p(\tau(s)) & : \quad \Psi_p(\alpha(s)) = true \\ \sigma_p(\tau(s)) & : \quad else \end{cases}$$

where the function $\Psi_p(t)$ evaluates predicate p on tuple t. In case, $p = a_i\theta a_j$ is a structural predicate, Ψ_p has the following semantics: Depending on θ, the predicate evaluates the binary structural relation $\uparrow$ (is parent of), $\downarrow$ (is child of), $\Uparrow$ (is ancestor of), $\Updownarrow$ (is ancestor or self of), $\Downarrow$ (is descendant of), $\Updownarrow$ (is descendant or self of), $\leftarrow$ (is preceding sibling of) $\rightarrow$ (is following sibling of), $\Leftarrow$ (is preceding of), $\Rightarrow$ (is following of), @ (is attribute of), and $\circlearrowleft$ (is self of). A structural predicate is evaluated to $\Psi_{a_i\theta a_j}(t) := t.a_i\theta t.a_j$. Note, if we want to express that "b is child of a" we write $b \downarrow a$ and not $a \downarrow b$. The order is important when we define the structural join. For the evaluation of the structural selection, an XML node identification mechanism (labeling scheme) is beneficial that can decide the relationship in question without a physical node access. Almost all XML database systems nowadays embody such a mechanism (e. g., OrdPath [24], DLN [2], SPLID [13], Pre/Post Numbering [3], etc.). In case of all other shapes of the predicate p, we refer to the original definition of the selection operator in [4].

Structural Join. With the help of the Cartesian product ($\times$) and the selection operator (σ_p), we define the join operator in the classic way:

$$s_1 \bowtie_p s_2 := \sigma_p(s_1 \times s_2)$$

This operator becomes a structural join operator when the join predicate checks structural relationships over attributes of the participating tuples. Note, the semantics of the structural join depends on the semantics of the ordered product ($\times$). Therefore, the output order of the structural join is significant. The structural semi-join ($\ltimes_p$), the structural anti-join ($\rhd_p$), and the structural left-outer join ($\bowtie_p$) are defined accordingly.

An interesting issue arises when thinking about the role of this new operator in NAL: Is the structural join a *logical* operator? To answer this question, we first have to state that the distinction between logical and physical operators in XML algebras is not as clear as in the relational world. Because *order* matters in XML, logical operators are defined in a way, respecting the requirement of order (like $\times$). But then, there is often only one chance to implement a logical operator, because other alternatives do not deliver the correct output order. Therefore, there is often no distinction between a logical operator and its physical implementation. However, for the structural join operator defined above, there are a lot of very efficient physical algorithms present. We even think that the combination of a D-Join with an unnest map operator is a physical implementation of the strutural join

defined above. Despite the intrinsic nested loop characteristics, we think our new operator qualifies as a logical one.

Node-Sequence Access. For the access to sequences of nodes having, for example, the same element name, we define the auxiliary function φ_p. For simplicity, its semantics is described in prose: $\varphi_p(c)$ is a function depending on the current evaluation context c. In the following, we omit context-parameter c for simplicity. It returns all nodes of a document in document order that comply with the predicate. For its evaluation, the function reads the current context node *cn* defined in the evaluation context, and calculates *cn*'s document root node. Then it scans the document in document order, thereby evaluating predicate p against each visited XML node. All qualifying nodes are returned in one sequence. In the following, φ_p will be used in combination with the Υ operator. For example, the expression $e = \Upsilon_{c:\varphi_{author}}(\Box)$ returns a sequence with $A(e) = c$ and all *author* elements in the current document as values. This functionality is required to provide bulk access to node sequences for the structural join operator, as similarly described in [6].

Nest. In the following, we will not need the complex grouping capabilities of the general unary/binary grouping operator provided in NAL [21]. A simple nesting and a simple grouping operator will do. Nesting is the complementary operator to unnesting. We assume the grouping operator in [21] to be defined on sets (or, more specifically, on vectors) of attributes A. Then, nesting is a shorthand for $\nu_{g:A}(e) = \Gamma_{g;=A;id}(e)$. If we want to nest by all attributes but the ones given in the vector A, we use $\nu_{g:\overline{A}}(e) = \Gamma_{g;=A(e)\setminus A;id}(e)$. Grouping will only occur in the form of unary, equality-based grouping, i. e., $\Gamma_{g;=A;f}(e)$.

Reverse, Group Reverse, and Group Sort. The reverse operator $\mathfrak{R}$ simply reverses the order of the tuples in the input sequence. If given an attribute name as subscript, $\mathfrak{R}_g$ assumes attribute g to be sequence valued. Then, it reverses the order of g's sequence. The group reverse operator $\mathfrak{R}^{\mathrm{G}}_A$ first nests its input by the attribute list A, reverses the order in each nesting group, and finally unnests the sequence again[9]:

$$\mathfrak{R}^{\mathrm{G}}_A(e_1) = \mu_g \circ \mathfrak{R}_g \circ \nu_{g:A}(e_1)$$

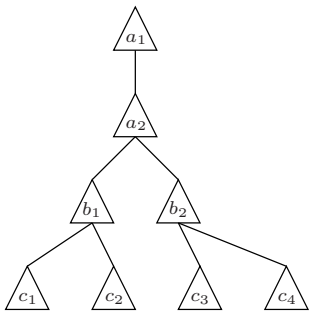
The same can be defined for the sort operator. Similarly to $Sort_{cn}$, the operator $\mathfrak{S}_g$ sorts the sequence valued g in ascending (document) order on the context node (*cn*). Then group-based sorting can be defined as:

$$\mathfrak{S}^{G}_A(e_1) = \mu_g \circ \mathfrak{S}_g \circ \nu_{g:A}(e_1)$$

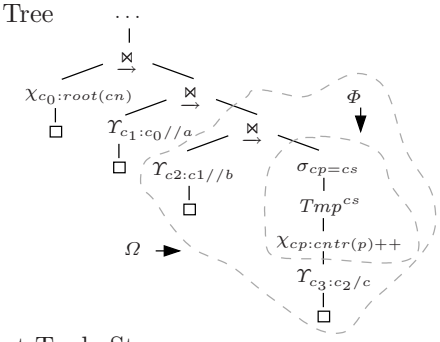
[9] Note, in contrast to [4] and [20] we do not use the "heavy" symbols $\mathfrak{R}$, $\mathfrak{A}$, and $\mathfrak{S}$ here. Instead, we employ the more convenient symbols $\mathfrak{R}$, $\mathfrak{A}$, and $\mathfrak{S}$.

a) Query $//a//b/c[pos() = last()]$

b) Document



c) NAL Tree



d) Correct Tuple Stream

c_{i-2}	c_{i-1}	c_i	cs	cp
a_1	b_1	c_1	2	1
a_1	b_1	c_2	2	2
a_1	b_2	c_3	2	1
a_1	b_2	c_4	2	2
a_2	b_1	c_1	2	1
a_2	b_1	c_2	2	2
a_2	b_2	c_3	2	1
a_2	b_2	c_4	2	2

Fig. 2 Query, Document, NAL Tree, and Tuple Stream

Sequence Merge and Intersect. The operators $\cup$ and $\cap$ are defined as the union and intersection of tuple sequences having the same schema.

Context Position and Context Size Calculation. The modifications on these two operators would have been discussed best after the next section. However, logically they belong here. Therefore, you may skip this part and return to this point after Section 4.

In the Natix Algebra, the special function $cntr(p_k)$++ is used in the subscript of a map operator (χ) to calculate the current content position (*cp*) for positional predicate p_k. Likewise, the Tmp^{cs} operator calculates the context size (*cs*). In the stacked translation semantics [4] and also for the replacement of the structural join [20], these operators are redefined to recognize group boundaries. The Tmp^{cs} operator is redefined to a Tmp^{cs}_c operator with the following semantics: $Tmp^{cs}_c = e\Gamma_{cs;c=c';count}\Pi_{c':c}(e)$. According to [4], this operator is used in the same way as the original operator, except that it is parameterized

with the input context node c_{i-1}, i. e., as $Tmp^{cs}_{c_{i-1}}$. Furthermore, the function calculating the current context position has to be altered for stacked translation. Reference [4] states that the function's internal counter has to be set to zero, when a new group is detected, i. e., when input context node (c_{i-1}) changes its value. Unfortunately, both operator semantics may lead to wrong results. As an example, consider Fig. 2. During the evaluation of the query in stacked translation mode, a tuple stream as shown in the first three columns (c_{i-2} to c_i) of the table is generated for the given query and document. For this stream, the context position and the context size have to be calculated. The correct values are shown in the remaining two columns of the table. However, the $Tmp^{cs}_{c_{i-1}}$ operator as defined in [4] and [20] would return a wrong result in this case, because the binary grouping by c_{i-1} would compute group size of 4 instead of 2. A counter example for the *count* method can be constructed from the above example, too: when subtree rooted at b_2 would be absent, no group boundary could be detected on attribute c_{i-1} (because this attribute would always contain the value b_1, although 2 groups are present).

Both operators are also required in this article for the introduction of the structural join operator to substitute D-Joins. To remedy these problems, we redefine them. Group-boundary detection is now based on a set of attributes A instead of a single one:

$$Tmp^{cs}_{A}(e) = e\Gamma_{cs;A=A';count}\Pi_{A':A}(e)$$

For the context position calculation, we substitute the *count* function with a $count^A$ function, which resets its internal counter based on a group-boundary detection regarding all attributes in A.

4 Introducing the Structural Join into a NAL Expression

In this section, we want to get rid of those D-Join operators that were initially compiled into a NAL expression for the calculation of path steps. With the following rewriting rule, the inherent node-at-a-time processing introduced by the D-Join is replaced by a set-at-a-time processing strategy which relies on the structural join.

$$e_j \left\langle \Phi \circ \Upsilon_{c_i:c_j/a_i::t_i}(\Box) \right\rangle = \Phi'(e_j \bowtie_{c_i\theta_{a_i}c_j} \Upsilon_{c_i:\varphi_{t_i}}(\Box)) \quad (1)$$

This rule has to be applied in an iterative fashion, thus substituting one D-Join operator at a time. Note, a direct compilation from XPath to NAL$^{\text{STJ}}$ is also possible. However, in this article we chose a given NAL expression as the starting point, because we want to ensure the equality of the resulting NAL$^{\text{STJ}}$ expression. After each application of the given rule, the resulting operator tree can still be evaluated, because NAL$^{\text{STJ}}$ is an extension of NAL.

The rule's pattern expression at the left-hand side is subject of the first lemma:

Lemma 1 *Let Ω be a NAL^{STJ} expression resulting from the XPath-to-NAL translation process described in [4] and a sequence of zero or more applications of Rule (1). If Ω (still) contains the calculation of a path step based on a D-Join, then Ω contains a subexpression $e_j \left\langle \Phi \circ \Upsilon_{c_i:c_j/a_i::t_i}(\Box) \right\rangle$, with the following properties:*

- *free variable c_j of the D-Join's inner expression is defined in its outer expression e_j,*
- *subexpression Φ does not contain any D-Join,*
- *subexpression Φ is maximal, i. e., Φ is not contained in any other NAL operator in the dependent part of the D-Join.*

Proof (Sketch): This lemma can be shown by an induction over the set of all XPath-to-NAL compilation rules and Rule (1). In each step, the existence of a Φ with the given properties can be shown. □

In other words, in the rule's pattern expression at the left-hand side, outer expression e_j generates a sequence of tuples containing an attribute c_j. For each tuple t, this attribute is the starting point for the calculation of the axis step in the dependent unnest-map expression. The result of the axis step is then processed by some subexpression Φ before the product with the tuple t is computed.

At the right-hand side of Rule (1), expression e_j is shifted into Φ', forming a structural join using the specified axis with a node sequence access $\Upsilon_{c_i:\varphi_{t_i}}$. This has the effect that Φ' consumes a slightly different input sequence, because it now contains also attributes from e_j. As an example, consider the query tree depicted in Figure 2c, which is evaluated on the given document. Here Φ is a cascade of operators, containing a selection, a context size, and a context position operator. During query evaluation, Φ is repeatedly executed on sequences of c elements, each sequence of which contains the children of a particular b element (e. g., $\langle c_1, c_2\rangle$ and $\langle c_3, c_4\rangle$). In this way, the groups of siblings are neatly separated. If we now want to employ a structural join instead of a D-Join for the evaluation of subtree Ω, Φ operates on a sequence of b–c tuples (e. g., $\langle\langle b_1, c_1\rangle, \langle b_1, c_2\rangle, \ldots, \langle b_2, c_4\rangle\rangle$). Therefore, Φ has to be transformed into expression Φ' to recognize the now implicit group-boundaries correctly. This transformation is defined by mapping function T on the operators contained in Φ. Function T will be identified by enumerating all mappings of interest. The correctness of the corresponding instances of Rule (1) will be shown.

Definition 1 Table 2 defines function T that maps a NAL$^{\text{STJ}}$ operator symbol onto another NAL$^{\text{STJ}}$ operator symbol. T depends on the input symbol Ψ, the outer expression e_j, and the axis direction of the axis calculated by the inner expression Υ as identified in Lemma

Table 2 Definition of NAL$^{\text{STJ}}$ Translation Function T

tuple-based ops.	**group-based operators**	**NAL$^{\text{STJ}}$ operators**
$T[id] = id$	$T[Tmp^{cs}, e_j] = Tmp^{cs}_{A(e_j)}$	$T[Tmp^{cs}_A, e_j] = Tmp^{cs}_{A \cup A(e_j)}$
$T[\sigma_p] = \sigma_p$	$T[\chi_{cn:cntr(p)++}, e_j, \overrightarrow{a}] = \chi_{cn:cntr^{A(e_j)}(p)++}$	$T[\chi_{cp:cntr^A(p)++}, e_j] = \chi_{cp:cntr^{A \cup A(e_j)}(p)++}$
$T[\chi_{cn:c_i}] = \chi_{cn:c_i}$	$T[\chi_{cn:cntr(p)++}, e_j, \overleftarrow{a}] = \mathfrak{R}^G_{A(e_j)} \circ \chi_{cn:cntr^{A(e_j)}(p)++} \circ \mathfrak{R}^G_{A(e_j)}$	$T[\nu_{g:A}, e_j] = \nu_{g:A \cup A(e_j)}$
$T[\mu_g] = \mu_g$	$T[\mathfrak{A}_{x:f}, e_j] = \Gamma_{x:=A(e_j);f}$	$T[\mathfrak{R}^G_A, e_j] = \mathfrak{R}^G_{A \cup A(e_j)}$
$T[\Pi_A] = \Pi_A$	$T[\Gamma_{x:=A;f}, e_j] = \Gamma_{x:=A \cup A(e_j);f}$	$T[\mathfrak{S}^G_A, e_j] = \mathfrak{S}^G_{A \cup A(e_j)}$
	$T[Sort_{cn}, e_j] = \mu_g \circ \mathfrak{S}^G_g \circ \nu_{g:A(e_j)}$	$T[\bowtie_p e_k] = \bowtie_p e_k$

1. Symbol $\overrightarrow{a}$ is used for forward axes and $\overleftarrow{a}$ for backward axes[10].

Theorem 1 *Let subexpression $\Phi = \Phi_1 \circ \cdots \circ \Phi_n$ be a NAL^{STJ} expression as defined in Lemma 1 and no Φ_i is an aggregation operator*[11]*. Then, Rule (1) is correct for $\Phi' = T[\Phi_1] \circ \cdots \circ T[\Phi_n]$.*

Proof : The correctness of the theorem relies on two properties: 1) the possibility to extract subexpression Φ out of the D-Join, thus generating Φ'—i. e., the correctness of all instances of the more general equality $e_1 \langle \Phi \circ e_2 \rangle = \Phi'(e_1 \langle e_2 \rangle)$—and 2) the equivalence of the resulting D-Join with the structural join as defined in (1), i. e., $e_j \left\langle \Upsilon_{c_i:c_j/a_i::t_i}(\square) \right\rangle = e_j \bowtie_{c_i \theta_{a_i} c_j} \Upsilon_{c_i:\varphi_{t_i}}(\square)$. The latter equality is clear from the definition of the participating operators: On the left-hand side, for each e_j the axis step is evaluated and the resulting intermediate sequence is joined with the input tuple. The right-hand side computes the same result employing an ordered cross product and a subsequent selection, which filters all tuples that do not conform to the axis specification.

For the proof of property 1), we decompose Φ into a sequence of operators, i. e., $\Phi = \Phi_1 \circ \cdots \circ \Phi_n$ and extract each single operator $\Phi_i = \Psi$ out of the D-Join, i. e., we show $e_1 \langle \Psi \circ e_2 \rangle = \Psi'(e_1 \langle e_2 \rangle)$. The operators Ψ in question can be divided into three classes:

- *tuple based*: operators that require one tuple at a time to compute the result for that tuple, for example, the selection operator σ which decides on each single tuple in the input sequence.
- *group based*: operators that require to process a group of tuples which belong to the same evaluation context, for example, the sort operator *Sort* which needs to "know" the complete group of tuples to be sorted.
- *NAL^{STJ}*: operators that are introduced by some mapping rule defined for one of the other two groups, i. e., that were introduced by an application of Rule (1).

For the proofs of all operators rewritings, we define a family of auxiliary functions $\alpha_{\langle i \rangle}$ on a sequence e, where a_i returns the i-th element. With α_i we can rewrite the D-Join in the above term:

$$e_1 \langle \Psi \circ e_2 \rangle = \bigoplus_{i=1}^{n} \alpha_i(e_1) \overline{\times} \Psi\Big(e_2(\alpha_i(e_1))\Big)$$

Our goal is now to extract the inner expression Ψ. We will distinguish the different operator groups:

- Because tuple-based operators only process one tuple at a time, it is easy to see that this processing can be "deferred" after the product with $\alpha_i(e_1)$ is computed, and, furthermore, after the intermediate tuple sequences (generated for each term) are concatenated. This means, we can transform the above equality to

$$\bigoplus_{i=1}^{n} \Psi\Big(\alpha_i(e_1) \overline{\times} \big(e_2(\alpha_i(e_1))\big)\Big) = \Psi\Big(\bigoplus_{i=1}^{n} \big(\alpha_i(e_1) \overline{\times} (e_2(\alpha_i(e_1)))\big)\Big) = \Psi(e_1 \langle e_2 \rangle)$$

- For group-based operators the situation is slightly different. As above, we can extract Ψ out of the inner expression, because the intermediate results after the product with $\alpha_i(e_1)$ has been computed have the same properties as before (i. e., the same size, order, etc.). But then, when Ψ has to be extracted out of the concatenation, the group boundaries do not exist anymore. Therefore, they have to be re-established by an additional nest operator. We get

$$\bigoplus_{i=1}^{n} \Psi\Big(\alpha_i(e_1) \overline{\times} \big(e_2(\alpha_i(e_1))\big)\Big) = \mu_g \circ \Psi_{\$g} \circ \nu_{g:A(e_1)} \circ \bigoplus_{i=1}^{n} \Big(\alpha_i(e_1) \overline{\times} \big(e_2(\alpha_i(e_1))\big)\Big) = \mu_g \circ \Psi_{\$g} \circ \nu_{g:A(e_1)}(e_1 \langle e_2 \rangle)$$

  The notation $\Psi_{\$g}$ indicates that this operator is now evaluated on the group bound by attribute g. This transformation is only possible, when expression e_1 does not produce any duplicates, because otherwise it would not be possible to distinguish different groups (produced for a duplicate tuple of e_1) properly. This

[10] When a particular mapping does not depend on e_j and a, those arguments are simply omitted.

[11] This circumstance will be discussed at the end of this section.

is also the reason, why it was not sufficient for Tmp_c^{cs} to check for variable binding c_{i-1} only. All rewritings presented in Table 2 for group-based operators are specializations of the term shown above. Note, backward axes together with the *cp* counter are rewritten using the group reverse operator $\mathfrak{R}_A^G$. This is necessary to comply with the XPath semantics.

- Of the NAL$^{\text{STJ}}$ operators, all but one are group-based. They are simply rewritten to recognize the "new" group boundaries introduced by e_j. The structural join operator $\bowtie e_k$ (where, according to the definition of Φ, e_k may not contain any D-Join operators) is not group-based. Its rewriting is equal to the one given for tuple-based operators. □

There is one circumstance related to the aggregation operator, in which the application of Rule (1) would not lead to a correct result. Therefore, aggregations are excluded in Theorem 1. For an example assume query $//a/count(b)$ which returns a sequence of integers, each of which represents the number of children under an a element. If we used Rule (1) on the NAL representation of this query, we would form a structural join between a and b elements and later on evaluate the aggregation. This, however, would lead to a wrong result, because the structural join discards those a elements for which no child b can be found, i. e., for which the answer of the aggregation should be zero. As a solution for this problem, we partition the aggregation functions into two groups: a) functions that generate a meaningful answer on an empty input (like, $count, exists, \ldots$), and b) all others (e. g., $min, max, \ldots$). If an aggregation based on one of the functions of group a) occurs in Φ, we alter Rule (1) to

$$e_j \left\langle \Phi \circ \Upsilon_{c_i:c_j/a_i::t_i}(\Box) \right\rangle = \Phi'(e_j \rtimes_{c_i \theta_{a_i} c_j} \Upsilon_{c_i:\varphi_{t_i}}(\Box)) \quad (2)$$

For all other aggregations, Rule (1) is correct. The applicability of Rule (1) is another interesting question, which is subject to the next theorem:

Theorem 2 *All D-Join operators initially compiled into a NAL expression Ω for the calculation of a path step as described in [4] can be substituted by a structural join as defined by Rule (1).*

Proof : Follows from Lemma 1. □

We conclude this section with the rewriting of a simplified version of the previous example: $/child :: a/child :: b\,[position() = last()]/child :: c$ (Figure 3). In the first step, e_1 and the depending subexpression can be identified as depicted in Figure 3a. Φ is the identity function, therefore, only the D-Join symbol is replaced. In Figure 3b, Φ contains a structural join (where the right branch does not contain any D-Join operator), a selection, a Tmp^{cs}, and a map operator. Here, the according transformations as defined in Table 2 are processed one at a time, resulting in Figure 3c. Although, the position-handling operators have already been made group aware

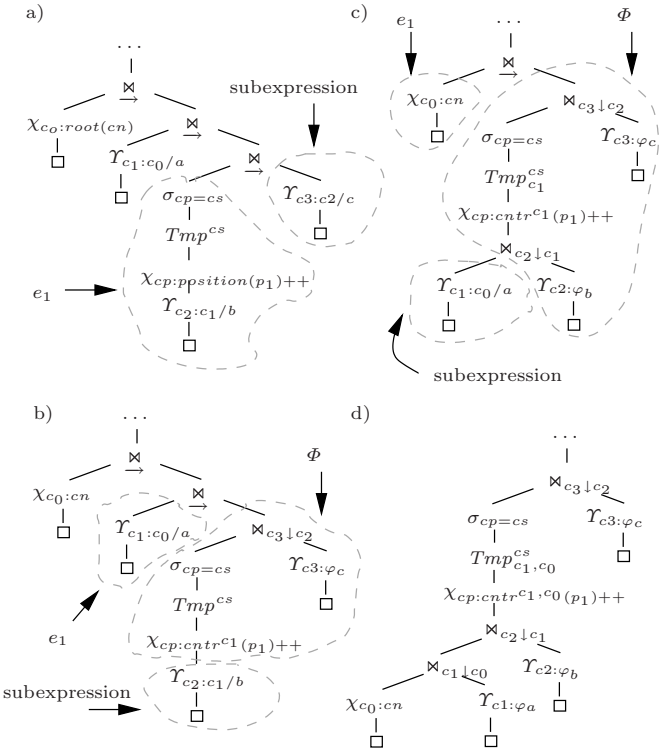


Fig. 3 Translation Example

in the previous step, in the last step, they have to be altered to recognize the new variable c_0. Note, of course, there are situations in which we can detect group boundaries without regarding *all* indicated variable bindings. However, the recognition of such situations is out of the scope of this paper.

5 Query Unnesting

With the introduction of the structural join into a NAL expression, we abandoned the explicit node-at-a-time path processing inherent to the D-Join operator. But still, the implicit node-at-a-time processing resulting from the evaluation of path steps in selections is present. In this section we will provide a set of unnesting rules to "expose" these hidden path step evaluations. We do not claim to have found all interesting rewritings possible, but we think, we cover the most common cases.

In this section, we will introduce unnesting rules for existential, conjunctive, disjunctive, and negated predicates. Furthermore, we will consider predicates based on aggregate functions. In all nested expressions, we assume relative path expressions to be present[12]. Our query unnesting strategies are not covered by the rules in [5, 21].

[12] Selections without nested path expressions are considered to be constant or positional.

Both contributions do not base their rewritings on the structural join operator.

5.1 Single-Path Predicates.

In the canonical XPath-to-NAL translation, path predicates are mapped onto selection operators. For an example see Figure 1. Relative path predicates require a pair of map operators ($\chi_{cn:c_i}$, $\chi_{c_0:cn}$) to "glue" the context of the outer expression to the subscript of a selection. The inner map is the starting point for a cascade of operators, the first one of which is a structural join (in NAL$^{\text{STJ}}$). Our goal is to "extract" the inner path expression and join it with the outer expression, thus abandoning the inherent node-at-a-time processing introduced by the selection operator. In some cases, we can replace the select operator completely. In other cases, we have to adjust the subscript to the new situation, using variable references to access necessary information, now produced in the outer expression. In the simple case, when the XPath predicate (and accordingly the selection subscript) contains only one relative path expression, we use the following generic unnesting rule:

$$\begin{aligned}&\sigma_{\Phi(\pi(\chi_{c_0:cn}(\Box)))}(\chi_{cn:c_0}(e_0)) = \\ &\Pi_{A(e_0)} \circ \sigma_{\Phi(\$g)} \circ \nu_{g:A(e_0)}(e_0 \rtimes_{c_1\theta c_0} \pi')\end{aligned} \tag{3}$$

At the left-hand side of this rule, you can find the above mentioned pair of map operators: The outer expression e_0 binds attribute c_0, which is then mapped onto *cn*; in the inner expression, e_0 is re-established from the context attribute *cn*. Variable π is a NAL$^{\text{STJ}}$ path expression depending on the context node given by the outer expression, i. e.,

$$\pi(\chi_{c_0:cn}(\Box)) = ((\chi_{c_0:cn}(\Box) \ltimes_{c_1\theta_1 c_0} e_1) \ldots \ltimes_{c_n\theta_n c_0} e_n)$$

In the following, we will simply abbreviate $\pi(\chi_{c_0:cn}(\Box))$ occurring in a selection subscript by π^χ. Φ is—as in the previous rewriting rules—a sequence of NAL$^{\text{STJ}}$ operators, but this time, it may not be the identity function. At the right-hand side we find a modified π'. The inner path expression π is extracted and joined with the outer e_0, using attribute c_1 of π in the join condition. Note, there is no need for map operators anymore, i. e., π' does not depend on $\chi_{c_0:cn}(\Box)$. This means that π' now has the form

$$\pi' = ((e_1 \ltimes_{c_2\theta_1 c_1} e_2) \ldots \ltimes_{c_n\theta_n c_0} e_n)$$

We denote this circumstance by the omission of the argument of π'. In the general case, the join operator has to be a left-outer join ($\rtimes$), because, as in the previous section, for some predicates Φ, the non-existence of a path π is of importance. To handle different evaluation contexts, a nest operator is inserted, which groups by all attributes, except those of π'. The selection is now executed on the grouped π', referencing the group by the variable $\$g$. After the selection, no information about the path π' is required anymore. Therefore, it is projected out. We will now show the correctness of the rewriting.

Theorem 3 *For a relative path expression π depending on the local context provided by the two map operators $\chi_{cn:c_0}$ and $\chi_{c_0:cn}$ for outer expression e_0 and the NAL^{STJ} predicate Φ, the rewriting presented in Rule (3) is correct.*

Proof : We first rewrite the selection operator in the following way: $\sigma_p(e) = \Pi_{A(e)} \circ \sigma_{x=\top} \circ \chi_{x:p}(e)$. On the right-hand side, the result of the predicate's evaluation has been made explicit: the map operator calculates the predicate on each tuple of input e and binds the result ($\top$ for "true", and $\bot$ for "false") onto a new attribute x. The intermediate sequence is then filtered by a "simpler" selection operator and, finally, the additional attribute x is projected out. With the definition of the map operator, we can now write:

$$\chi_{x:p}(e) = \bigoplus_{i=1}^{n} \alpha_i(e) \circ [x : p(\alpha_i(e)] = \bigoplus_{i=1}^{n} \alpha_i(e) \overline{\times} p_x(\alpha_i(e))$$

For the term on the right-hand side, we simply used the definition of the product operator ($\overline{\times}$) and "integrated" the construction of the new tuple (binding x) into the predicate, denoted by p_x[13]. This expression is obviously a D-Join. Therefore, we can write:

$$\sigma_p(e) = \Pi_{A(e)} \circ \sigma_{x=\top}(e \langle p_x \rangle)$$

We now apply this rewriting to the left-hand side of (3):

$$\begin{aligned}&\sigma_{\Phi(\pi^\chi)}(\chi_{cn:c_0}(e_0)) = \\ &\Pi_{A(e_0)} \circ \sigma_{x=\top}\Big(\chi_{cn:c_0}(e_0) \langle \Phi_x(\pi^\chi) \rangle\Big)\end{aligned}$$

The same question as in the previous section arises: How can Φ_x be extracted and how can the D-Join be replaced by a structural join? From [4] we learn, that predicates containing relative paths are translated using an aggregation operator. For example the predicate in $a[.//b]$ is translated using a $\mathfrak{A}_{x:exists}$ operator. Because the above expression has the same structure as the expressions shown in the proof for Rule (1) and (2) (except for the explicit context node mapping), we can use those rewritings here. With Φ_x being identified as *group based*, we can write:

$$\begin{aligned}&\Pi_{A(e_0)} \circ \sigma_{x=\top}\Big(\chi_{cn:c_0}(e_0) \langle \Phi_x(\pi^\chi) \rangle\Big) = \\ &\Pi_{A(e_0)} \circ \sigma_{x=\top} \circ \Phi_{x:\$g} \circ \nu_{g:A(e_0)}(e_0 \rtimes_{c_1\theta c_0} \pi') = \\ &\Pi_{A(e_0)} \circ \sigma_{\Phi(\$g)} \circ \nu_{g:A(e_0)}(e_0 \rtimes_{c_1\theta c_0} \pi')\end{aligned}$$

In the first step, Φ_x is extracted and a left-outer structural join is formed on e_0 and π. For this structural join,

[13] In many cases, this happens anyway, e. g., when a $\mathfrak{A}_{x:exists}$ aggregation is evaluated in the selection subscript.

the map operators are not required anymore. If predicate Φ does not produce any input for non-existent paths (i. e., for an empty input, as before), we can safely use an ordinary structural join ($\bowtie$) here. Because the extracted Φ_x is group based, the proper groups have to be established by a nest operator. Φ evaluates now on the group variable $\$g$. In the second step, we merge Φ and the selection operator. □

While this rule is directly applicable, there are further refinements for special cases that result in a much simpler rewriting.

Unnesting Existential Predicates. Sometimes plain path predicates like in $a[b/c]$ occur. In NAL$^{\text{STJ}}$, those expressions are compiled to an aggregation in combination with an *exists* in the selection subscript. They can be unnested with the following rule, introducing a semi-join operator:

$$\sigma_{A_{x:exists}(\pi^\chi)}(\chi_{cn:c_0}(e_0)) = e_0 \ltimes_{c_1\theta c_0} \pi' \tag{4}$$

Note, on the right-hand side, π' is evaluated, before the structural join is computed. Essentially this means that π' is not evaluated in the context of e_0 anymore. This could be problematic, if π' returns a large number of intermediate tuples. Another solution is viable as well, where path expression π is exposed:

$$\begin{aligned}&\sigma_{A_{x:exists}(\pi^\chi)}(\chi_{cn:c_0}(e_0)) = \\ &\quad \Pi^D_{A(e_0)}((e_0 \bowtie_{c_1\theta c_0} e_1) \cdots \bowtie_{c_n\theta c_{n-1}} e_n)\end{aligned} \tag{5}$$

In the case of a negated path predicate, e. g., $a[not(b/c)]$, we use an anti-join operator:

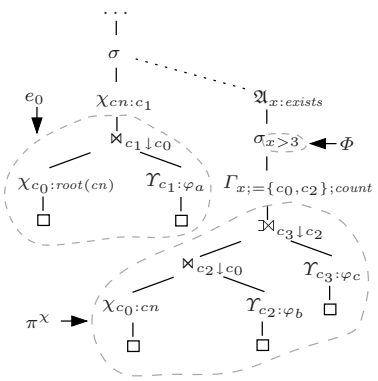
$$\sigma_{A_{x:\neg exists}(\pi^\chi)}(\chi_{cn:c_0}(e_0)) = e_0 \rhd_{c_1\theta c_0} \pi' \tag{6}$$

Unnesting Predicates with Aggregate Functions. If the nested sub-expression contains an aggregate function, e. g., as in $a[count(b) > 3]$, we can unnest this query using a group-by in combination with the aggregate function:

$$\begin{aligned}&\sigma_{\Phi(\mathfrak{A}_{x:f}(\pi^\chi))}(\chi_{cn:c_0}(e_0)) = \\ &\quad \Pi_{A(e_0)} \circ \sigma_{\Phi(\$x)} \circ \Gamma_{x;=A(e_0);f}(e_0 \mathbin{⟕}_{c_1\theta c_0} \pi')\end{aligned} \tag{7}$$

Note, the above depicted situation does not naturally occur in a NAL$^{\text{STJ}}$ expression, because an aggregation operator is translated to a grouping operator when a structural join is introduced (see previous section). However, this rule can be relaxed, when the aggregate function is applied to the *complete* path in the XPath predicate, as in the example above, retaining the aggregation operator. An example for this rule is contained in the final example at the end of this section. A more complicated situation occurs, when the aggregate function is applied to some sub-path, as in $a[b/count(c) > 3]$. Then, due to the introduction of the structural join, a group operator *has to* occur in the selection, instead of the simple

a) NAL$^{\text{STJ}}$ Expression



b) Unnested Expression

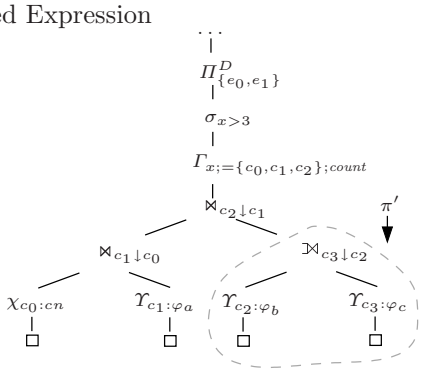


Fig. 4 Example for Rule (8) on query $a[b/count(c) > 3]$

aggregation operator. This make the following rewriting necessary:

$$\begin{aligned}&\sigma_{\mathfrak{A}_{x:exists} \circ \sigma_\Phi \circ \Gamma_{x;=A;f}(\pi^\chi)}(\chi_{cn:c_0}(e_0)) = \\ &\quad \Pi^D_{A(e_0)} \circ \sigma_\Phi \circ \Gamma_{x;=A\cup A(e_0);f}(e_0 \bowtie_{c_1\theta c_0} \pi')\end{aligned} \tag{8}$$

An example for this rather complicated rule is depicted in Fig. 4.
The correctness of Rules (4) to (8) can be shown in a similar fashion as the proof of Rule (3). Note, the left-outer join operators ($⟕$) in rewritings (3) and (7) are only required, if the participating operators (Φ and $\mathfrak{A}$) produce an output for an empty input sequence. As in the previous chapter, these left-outer joins can be substituted by ordinary joins, if this is not the case.

5.2 Multi-Path Predicates

In contrast to the previous rewriting rules, Φ may not be unary anymore, because in one predicate, several path expressions can be evaluated "simultaneously". This leads to a generic unnesting rule for the complex case, when multiple path expressions are located in a single attribute:

$$\begin{aligned}&\sigma_{\Phi(\pi_1^\chi,\ldots,\pi_n^\chi)}(\chi_{cn:c_0}(e_0)) = \\ &\quad \Pi_{A(e_0)} \circ \sigma_{\Phi(\$g_1,\ldots,\$g_n)} \circ \nu_{g_1:\overline{A(\pi_1')}} \circ \cdots \circ \nu_{g_n:\overline{A(\pi_n')}} \\ &\qquad ((e_0 \mathbin{⟕}_{c_1\theta c_0} \pi_1') \cdots \mathbin{⟕}_{c_n\theta c_0} (\pi_n'))\end{aligned} \tag{9}$$

Here, Φ is n-ary, depending on a set of path expressions. Because all path expressions are evaluated in the same local context, the depicted nesting is actually possible: no nesting of already nested sequences may occur. The only critical issue arising is the calculation of a nesting, where attributes compared for equality may be sequence valued. This is, however, not a problem of the logical algebra, but has to be solved at the physical level. One strategy, for example, would be to abandon the nest operators and modify the subsequent operators to make them *group aware*[14]. Another possible solution is to integrate the generation of nested groups into physical structural join operators, as sketched in [17]. The correctness of the above rule can be shown by an induction of the paths to be extracted. This proof works in a similar fashion as the one for Rule (3). As before, depending on Φ the left-outer join can be replaced by an ordinary one for certain π_i.

Again, we provide some rewriting rules for special cases in the following.

Rewriting Conjunctive Predicates. Whenever possible, we normalize the subscripts of selections into a disjunctive form, i. e., $e_1 \wedge (e_2 \vee e_3) = (e_1 \wedge e_2) \vee (e_1 \wedge e_3)$. We are aware that, by multiplying e_1, common subexpressions are introduced. Again, this is not a problem for the logical algebra, but the physical plan generator has to deal with it. Every time we have to introduce common subexpressions, we give the plan generator a hint to signal their correspondence.

The first rewriting handles conjunctive expressions. For them, we rewrite the query using the well-known equivalence:

$$\sigma_{e_2 \wedge e_3}(e_1) = \sigma_{e_2} \circ \sigma_{e_3}(e_1) = \sigma_{e_3} \circ \sigma_{e_2}(e_1) \tag{10}$$

Thus, the problem of multiple-path expressions is reduced to the problem of single-path expressions.

Rewriting Disjunctive Predicates. Disjunctive predicates may be handled similarly to conjunctive ones (i. e., reduction to single-path expressions) using the sequence merge operator:

$$\sigma_{e_2 \vee e_3}(e_1) = \sigma_{e_2}(e_1) \cup \sigma_{e_3}(e_1) = \sigma_{e_3}(e_1) \cup \sigma_{e_2}(e_1) \tag{11}$$

Again, this rewriting requires special care from the plan generator to handle the multiplied occurrences of expression e_1. When subexpressions of the disjunction are aggregated using the *exists*() function, they can be extracted by using left-outer joins:

$$\begin{aligned} &\sigma_{\mathfrak{A}_{x:exists}(\pi^\chi) \vee e_2}(\chi_{cn:c_0}(e_0)) = \\ &\qquad \Pi^D_{A(e_0)} \circ \sigma_{(A(\pi') \neq \epsilon) \vee e_2}(e_0 \bowtie_{c_1 \theta c_0} \pi') \end{aligned} \tag{12}$$

[14] This technique has already been applied in the stacked translation, where the Tmp^{cs} operator is converted to a group-aware $Tmp^{cs}_{c_i}$ operator

The notation $A(\pi') \neq \epsilon$ essentially has the meaning $\forall a \in A(\pi') : a \neq \epsilon$, i. e., π' has provided a join partner in the left-outer join. Of course, the map operator $\chi_{cn:c_0}$ can only be abandoned, if expression e_2 does not contain any relative path anymore. In the case, when multiple path expressions in a general disjunction may occur, the query can be rewritten as:

$$\begin{aligned} &\sigma_{\Phi_1(\pi_1^\chi) \vee \Phi_2(\pi_2^\chi)}(\chi_{cn:c_0}(e_0)) = \\ &\quad \Pi_{A(e_0)} \circ \sigma_{(\Phi_1(\$g_1) \vee \Phi_2(\$g_2))} \circ \\ &\quad \nu_{g_1:\overline{A(\pi_1')}} \circ \nu_{g_2:\overline{A(\pi_2')}} \\ &\qquad ((e_0 \bowtie_{c_1 \theta c_0} \pi_1') \bowtie_{c_2 \theta c_0} \pi_2') \end{aligned} \tag{13}$$

In Rule (12) and (13), the left-outer join cannot be replaced by an ordinary join, because then we would accidentally "throw away" intermediate results. For example, by using an ordinary join between a and b for expression $a[b \vee c]$, we would miss all a elements which should be part of the final result due to c.

Unnesting Path Comparison Expressions. In the NAL compilation process, predicates of the form $[e_1 \theta e_2]$ are translated into an $\mathfrak{A}_{x:exists}$ predicate. Therefore, with Rule (4), we can also unnest predicates that contain a comparison of a path with a constant (*simple* path comparison expression). For example, the query $a[b > 3]$ can be translated and unnested into the NAL$^{\text{STJ}}$ expression[15]

$$\Pi^D\Big(\chi_{cn:c_1}\big((\chi_{c_0:cn} \ltimes_{c_1 \downarrow c_0} \Upsilon_{c_1:\varphi_a}) \ltimes_{c_2 \downarrow c_1} (\sigma_{>3}(\Upsilon_{c_2:\varphi_b}))\big)\Big)$$

However, because Φ is unary, this rewriting rule does not provide any help in case of *complex* path comparison expressions like $a[b/text() = c/text()]$. In such a case, the following unnesting rule can be applied.

$$\begin{aligned} &\sigma_{\mathfrak{A}_{x:exists} \circ \Phi_\theta(\pi_1^\chi, \pi_2^\chi))}(\chi_{cn:c_0}(e_0)) = \\ &\qquad \Pi^D_{A(e_0)} \circ \sigma_{(\$c_1 \theta \$c_2)}((e_0 \ltimes_{c_1 \theta c_0} \pi_1') \ltimes_{c_2 \theta c_0} \pi_2') \end{aligned} \tag{14}$$

In this rule Φ_θ is the compilation of the existential comparison as introduced in [4]. For example $\pi_1 = \pi_2$ would be compiled into $\mathfrak{A}_{exists} \pi_1 \ltimes \pi_2$. Rule (13) is promising, because it may be implemented very efficiently. At the right-hand side, the selection operator simply compares two attributes. This comparison has non-existential semantics, in contrast the existential semantics on the left-hand side. The generated tuple stream is in document order. Therefore, the duplicate elimination operator is simply a buffered filter with a buffer size of one tuple.

[15] Because 3 is a constant, we do not compile it using an aggregation, e. g., $\mathfrak{A}_{max_{cn}(3)}$, as suggested in [4].

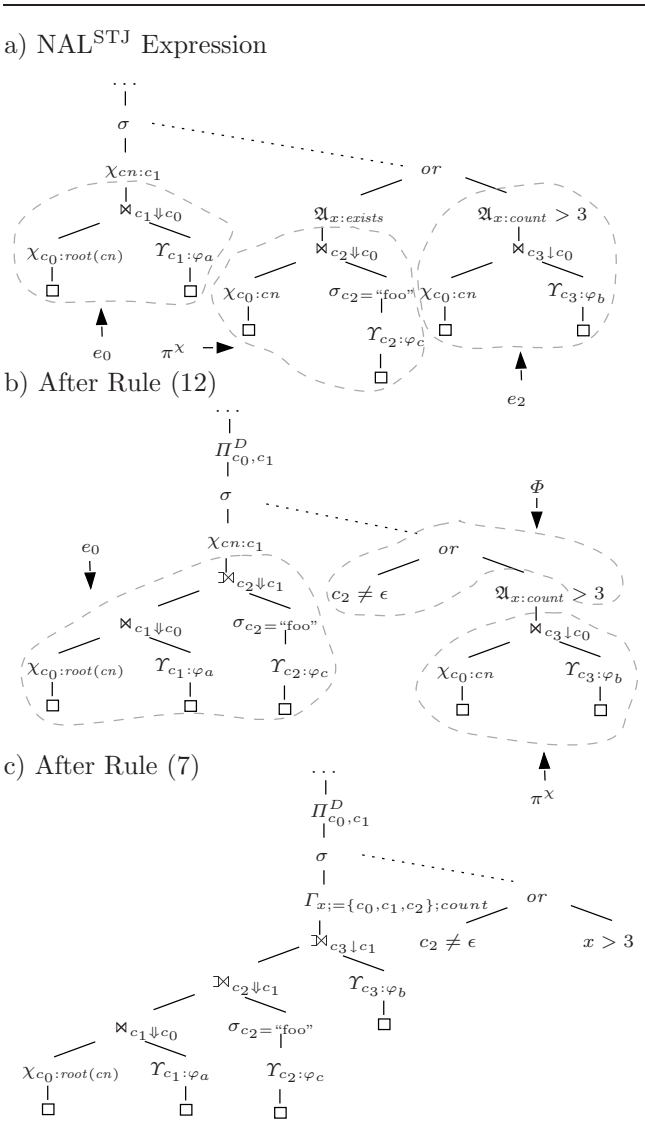


Fig. 5 Unnesting of query $//a[.//c = foo \vee count(b) > 3]$

An Unnesting Example. We will conclude the discussion of query unnesting with an example. Consider the XPath expression $/desc :: a[child :: c =$ "foo" $\vee\ count(desc :: b) > 3]$. Its NAL$^{\text{STJ}}$ compilation is presented in Figure 5a. To keep this example simple, we slightly deviated from the originally generated NAL$^{\text{STJ}}$ expression: instead of employing a semi-join operator for the compilation of the first XPath predicate (as suggested in [4]), we used a selection operator for the check of "foo". Furthermore, we use $\mathfrak{A}_{x:count} > 3$ as a predicate instead of a more complex expression relying on the group operator[16]. In the first step, Rule (12) can be applied. This results in a mandatory left-outer join between e_0 and π^{χ}. In the second step, we can apply Rule (7), because now the selection contains only one relative path expression. Rule (7) also introduces a left-outer join and, additionally, a group operator. Because the structural join operators are now "neighbors", it is easy to reorder the join operators, as described in [27]. Furthermore, the mapping onto a single holistic twig join operator is also straightforward.

Finally, a slight optimization can be pointed out: the check for "foo" can simply be integrated into the node sequence access ($\Upsilon_{\varphi_c \wedge \text{"foo"}}$). In a physical algebra, this type of access could be supported by an appropriate content and structure index.

[16] This is actually possible, because the aggregation is calculated for the complete path, as described in the discussion of Rule (7).

6 Pushing Down Structural Joins

The mapping of a logical algebra expression to its corresponding physical one is out of the scope of this paper. However, when thinking about this mapping, two interesting questions arise: How can a *logical* expression be "prepared" to facilitate the logical-to-physical algebra mapping and how can the problem of structural join order selection be tackled? We think the answer to these questions lies in a special operator tree format, where the tuple-generating structural joins are located at the bottom of the tree (as in Figure 5c), and filtering/selection operators occur as inner nodes. In this representation, logically related path processing operators are situated close to each other. Because no selections or other operators interfere, it is easy to determine the different parts to be mapped onto a holistic twig join operator, a path index access, or onto the structural join operator. This operator tree format can be generated by lifting non-structural join operators out of either side of a structural join operator. For example, in Figure 3d, from the left side of the final structural join, the selection, *Tmp*, and χ operators could be lifted, pushing down the structural join to the bottom of the tree. In Table 3 we provide rewriting rules to accomplish such restructurings. We are aware that these rules have an immediate impact on the costs of the query, because the evaluation of selections—minimizing the intermediate result size—is deferred. However, using the same set of rules, these selections may be pushed back into their original places, after the logical-to-physical mapping has been performed.

Because these equivalences may be read from either side, they also provide a way to push down non-structural operators. Again, we do not claim to have found all interesting rewritings possible here. In addition to the rules depicted in Table 3, we have found rules to push down a join over the special operators $Tmp^{cs}_{c_j}$ and $\chi_{cp:cntr(p)++}$. However, their discussion is beyond the scope of this paper.

To conclude the discussion about logical query rewriting, we point out some limitations that may provide a starting point for future research:

- Although developed in the context of XPath, the given equivalences may also be applied in algebra expressions for general XML queries ([21] shows an

Table 3 Join Push-Down Equivalences

Operator	Rule	Condition
σ_p (Selection)	$\sigma_p(e_1) \bowtie_{c_2\theta c_1} e_2 = \sigma_p(e_1 \bowtie_{c_2\theta c_1} e_2)$	$F(p) \cap A(e_2) = \emptyset$
Π_A (Projection)	$\Pi_A(e_1) \bowtie_{c_2\theta c_1} e_2 = \Pi_{A \cup A(e_2)}(e_1 \bowtie_{c_2\theta c_1} e_2)$	$c_1 \in A$
Π^D (Dup. Elim.)	$\Pi^D(e_1) \bowtie_{c_2\theta c_1} e_2 = \Pi^D(e_1 \bowtie_{c_2\theta c_1} e_2)$	e_2 duplicate free
$\Pi_{\overline{A}}$ (Projection)	$\Pi_{\overline{A}}(e_1) \bowtie_{c_2\theta c_1} e_2 = \Pi_{\overline{A}}(e_1 \bowtie_{c_2\theta c_1} e_2)$	$A \cap A(e_2) = \emptyset \wedge c_1 \notin A$
$\Gamma_{x;=A;f}$ (Group)	$\Gamma_{x;=A;f}(e_1) \bowtie_{c_2\theta c_1} e_2 = \Gamma_{x;=A\cup A(e_2);f}(e_1 \bowtie_{c_2\theta c_1} e_2)$	$c_1 \in A$
$\nu_{g:A}$ (Nest)	$\nu_{g:A}(e_1) \bowtie_{c_2\theta c_1} e_2 = \nu_{g:A\cup A(e_2)}(e_1 \bowtie_{c_2\theta c_1} e_2)$	$c_1 \in A$
μ_g (Unnest)	$\mu_g(e_1) \bowtie_{c_2\theta c_1} e_2 = \mu_g(e_1 \bowtie_{c_2\theta c_1} e_2)$	$c_1 \notin A(g)$

XQuery-to-NAL compilation). However, we do not provide any means to "find" twig patterns in an algorithmic expression, as [23] does.
- We do not claim to have found *all* possible rewritings.
- A "blind" application of the equivalences is not possible in general. Therefore, a concise cost model and document statistics are required.
- We did not provide any rules to reorder structural joins (see [27] for further research).
- We did not show how to generate XML query plans, i. e., how to actually employ the Holistic Twig Join and access to path indexes.

7 Quantitative Results

To substantiate our findings, we compared the different evaluation strategies by a one-to-one comparison on a single-user system. We implemented the operators of the NAL^{STJ} algebra in the XTC system. Because we wanted to keep the comparison between a pure NAL expression and the NAL^{STJ} variants of a query simple and, because we do not elaborate on a sophisticated logical-to-physical algebra mapping in this paper, we just used the algorithm presented in [1] for the implementation of the structural join.

System Testbed. XTC is one of the few native database systems providing fine-grained transaction isolation over shared XML documents. In XTC, each XML node has a unique stable path labeling identifier (SPLID [13]). We refined the ORDPATH [24] concept for the implementation of SPLIDs. For document storage, each node is mapped onto a record, containing the SPLID and the encoded node data. All records of a document are stored in a B*-Tree, comprising the *document container*.

Furthermore, the *element index* provides for fast access to elements with the same element name (see Figure 6). It is a two-way index, consisting of a name directory (B-Tree) and a set of node-reference indexes. Given a context node *cn*, the element index can be used to calculate the sequence of all elements having a specific

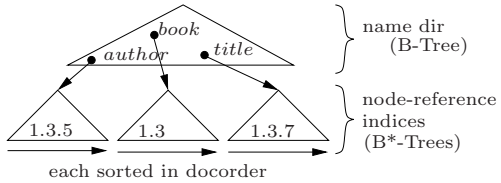


Fig. 6 Element Index

name on a specific axis. Such queries are simply translated to range queries over a particular node-reference index. This is exactly, how we implemented the evaluation of the Υ operator. XPath predicates subject to the value content of XML nodes are evaluated on the document index.

Query Workload. The query workload depicted in Table 4 was run on four XMark [26] documents of size 120 KB, 1.2 MB, 12 MB, and 112 MB (factors 0.001, 0.01, 0.1, 1). Each query was compiled into the pure NAL stacked translation and into its (optimized) unnested equivalent in NAL^{STJ}. To address various XPath use cases, we tested the following types of queries: a purely structural query, a query relying on position, a content-based query, and a query with aggregations. For the structural query, the NAL expression does not examine all dependent paths in the path predicate. When the first matching path is found, the evaluation of the predicate is accomplished.

Results. Our tests were carried out on an Intel XEON computer (four 1.5 GHz CPUs, 2 GB main memory, 300 GB external memory, Java Sun JDK 1.5.0) as the XDBMS server machine and a PC (1.4 GHz Pentium IV CPU, 512 MB main memory, JDK 1.5.0) as the client, connected via 100 MBit ethernet to the server. All tests were issued on a hot DB buffer of 250 16KB-sized pages.

Our first observation is that the figures of all queries look very similar. On the small document, both NAL and NAL^{STJ} show the same performance. However, as the documents and the result sizes grow larger, the NAL^{STJ} optimized expressions are roughly one magnitude faster

Table 4 Query Workload

No	Query	Characteristics
Q1	$//closed\_auction/annotation/description[parlist/listitem/text/keyword]$	purely structural
Q2	$//open\_auctions/open\_auction/bidder[position() = last() \vee position() = 1]$	positional
Q3	$//item[.//date = \text{“}20/07/2000\text{”} \wedge ./payment = \text{“}Creditcard\text{”}]$	content based
Q4	$//item[count(.//text//bold) > 5 \vee count(.//mail) > 3]$	aggregational

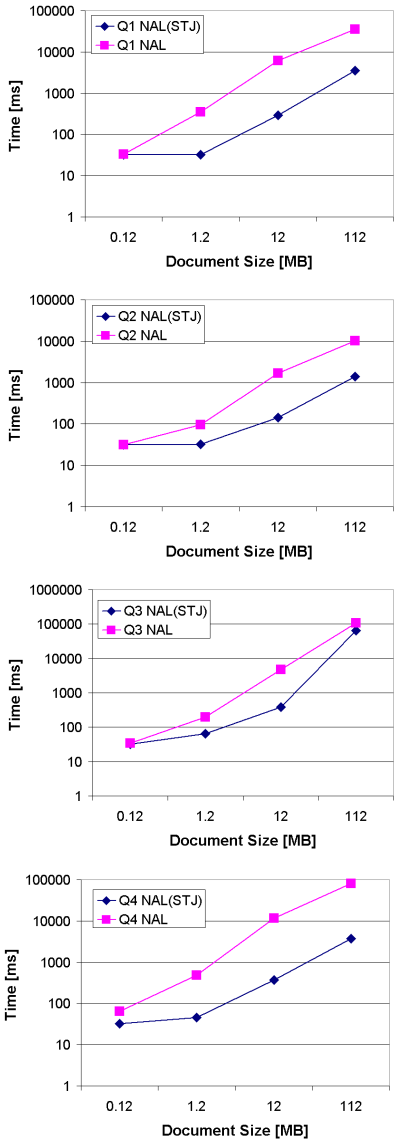


Fig. 7 Queries Q1, Q2, Q3, and Q4

(note, we used the log scale on the x-axis and the y-axis). The only exception is the content-based query Q3. Furthermore, we notice that both strategies scale with respect to the size of the input document.

The major explanation for the above effects is the relation between *node-at-a-time path* processing (D-Joins in NAL) and *set-at-a-time* path processing (structural joins in NAL$^{\text{STJ}}$). For example in NAL, query Q4 is evaluated by accessing all *item* elements and, for each such element, evaluating the predicate. This implies a repeated access to the element index to scan the depending predicate paths. In contrast to this, set-at-a-time requires only few element index scans which are carried out in a sequential fashion. On small documents where only a few intermediate tuples occur, the distinction between the two processing styles does not carry much weight. However, when the element index has to be accessed over and over again, e. g., due to a large input in a selection predicate, access costs explode.

The problem with query Q3 is that, for NAL$^{\text{STJ}}$, Q3 also requires node-at-a-time processing to evaluate the content predicate. This is due to the lack of a content index in the XTC system. If we could access text nodes carrying the same content as easily as element nodes with the same element name, then it would also be possible to evaluate the equality predicate using a structural join.

We are aware that all presented queries could be evaluated faster, if suitable measures on the mapping from the logical to the physical algebra were taken. For example, Q1 could be answered more easily with the help of a structural index, even if only a sub-path of the query could be evaluated by that index. For queries with positional predicates, special evaluation algorithms resembling structural joins have been proposed [29]. In query Q3, a text index, as sketched above, would be very beneficial. A structural join reordering could take the selectivity of the text predicate into account and start the evaluation by the computation of a structural join between the *date* elements and the value “07/05/2005”. However, to keep the two strategies comparable, we contented ourselves with the simple mapping sketched above.

8 Conclusions

To the best of our knowledge, this is the first article dealing with the introduction of the structural join operator into a tuple-based XPath algebra. With our contribution, we hope we can bridge the gap between the many promising algebra proposals on one side and the equally many proposals on evaluation algorithms (physical operators) for XML queries on the other side. We are aware

that this is only an initial step towards the integration of these valuable concepts, because many problems regarding the logical-to-physical algebra mapping are still left out, e. g., join reordering, cost-based optimization, etc.

With the structural join, it is now possible to substitute implicit (selections) and explicit (D-Join) node-at-a-time processing steps in operator plans. Note, this is accomplished at the logical level only; a physical implementation may freely choose to implement a query, predicate, or even only a structural join in a node-at-a-time manner (for the last point, see [18]), nevertheless. Even hash-based strategies may be applied [19]. The decision to do so depends on physical issues and cannot be determined at the logical level. However, we provided essential rewriting rules to ensure this flexibility.

Even with the given simple mapping from a logical algebra expression to a physical one (taking only the algorithm from [1] into account), we gained an order of magnitude in the performance of query evaluation.

Acknowledgements I would like to thank Theo Härder, Jose de Aguiar Moraes Filho, Andreas Weiner and the anonymous referees for their valuable comments on this paper. The support of Andreas Bühmann while formatting the final version is appreciated.

References

1. S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, D. Srivastava: Structural Joins: A Primitive for Efficient XML Query Pattern Matching. Proc. ICDE: 141–152 (2002)
2. T. Böhme, E. Rahm: Supporting Efficient Streaming and Insertion of XML Data in RDBMS. Proc. 3rd DIWeb Workshop: 70-81 (2004)
3. P. A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, J. Teubner: MonetDB/XQuery: a fast XQuery processor powered by a relational engine. Proc. SIGMOD: 479–490 (2006)
4. M. Brantner, C.-C. Kanne, S. Helmer, G.Moerkotte: Full-fledged Algebraic XPath Processing in Natix. Proc. ICDE: 705–716 (2005)
5. M Brantner, C.-C. Kanne, S. Helmer, G. Moerkotte: Algebraic Optimization of Nested XPath Expressions. Proc. ICDE: 128 (2006)
6. N. Bruno, N. Koudas, D. Srivastava: Holistic twig joins: Optimal XML pattern matching. Proc. SIGMOD: 310–321 (2002)
7. Q. Chen, A. Lim, K. W. Ong: D(k)-Index: An Adaptive Structural Summary for Graph-Structured Data. Proc. SIGMOD: 134–144 (2003)
8. S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, C. Zaniolo: Efficient Structural Joins on Indexed XML Documents. Proc. VLDB: 263–274 (2002)
9. W3C Recommendation: XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Specification. http://www.w3.org/TR/xquery-semantics/
10. M. F. Fernandez, J. Hidders, P. Michiels, J. Simeon, R. Vercammen: Optimizing Sorting and Duplicate Elimination. Proc DEXA: 554-563 (2005)
11. M. Fontoura, V. Josifovski, E. Shekita, B. Yang: Optimizing Cursor Movement in Holistic Twig Joins. Proc. 14th CIKM: 784–791 (2005)
12. T. Grust, M. van Keulen, J. Teubner: Staircase Join: Teach a Relational DBMS to Watch its Axis Steps. Proc VLDB: 524–535 (2003)
13. T. Härder, M. Haustein, C. Mathis, M. Wagner: Node Labeling Schemes for Dynamic XML Documents Reconsidered. Data & Knowledge Engineering 60: 126–149 (2007)
14. J. Hidders, P. Michiels, J. Siméon, and R. Vercammen: How to recognize different kinds of tree patterns from quite a long way away. Technical Report TR UA 13-2006, Univ. of Antwerp and IBM Research, 2006.
15. H.-G. Li, S. Alireza Aghili, A. El Abbadi: FLUX: Content and Structure Matching of XPath Queries with Range Predicates, Proc. XSym: 61–76 (2006)
16. H. Jagadish and L. Lakshmanan and D. Srivastava and K. Thompson: TAX: A Tree Algebra for XML. Proc. DBPL: 149–164 (2001)
17. S. Paparizos, Y. Wu, L. V. S. Lakshmanan, H. V. Jagadish: Tree Logical Classes for Efficient Evaluation of XQuery. Proc. SIGMOD: 71–82 (2004)
18. Mathis, Ch., Härder, T., Haustein, M.: Locking-Aware Structural Join Operators for XML Query Processing. Proc. SIGMOD: 467–478 (2006)
19. Mathis, Ch., Härder, T.: Hash-Based Structural Join Algorithms, Proc. DATAX'06, LNCS 4254, Springer-Verlag, 136–149 (2006)
20. Mathis, Ch.: Integrating Structural Joins into a Tuple-Based XPath Algebra, Proc. BTW: 242–261 (2007)
21. N. May, S. Helmer, G. Moerkotte: Nested Queries and Quantifiers in an Ordered Context. Proc. ICDE: 239–250 (2004)
22. N. May, M. Brantner, A. Böhm, C.-C. Kanne, G. Moerkotte: Index vs. Navigation in XPath Evaluation. Proc. XSym: 16–30 (2006)
23. P. Michiels, G. A. Mihaila, J. Siméon: Put a Tree Pattern in Your Algebra. Proc. ICDE (2007)
24. P. E. O'Neil, E. J. O'Neil, S. Pal, I. Cseri, G. Schaller, N. Westbury: ORDPATHs: Insert-friendly XML node labels. Proc. SIGMOD: 903–908 (2004)
25. C. Re, J. Siméon, M. Fernández: A Complete and Efficient Algebraic Compiler for XQuery. Proc. ICDE: 14 (2006)
26. A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, R. Busse: XMark: A Benchmark for XML Data Management. Proc. VLDB: 974–985 (2002)
27. Y. Wu, J. M. Patel, H. V. Jagadish: Structural Join Order Selection for XML Query Optimization. Proc. ICDE: 443–454 (2003)
28. W3C Recommendation: XML Path Language (XPath), Version 1.0 (1999). http://www.w3.org/TR/xpath
29. V. Zografoula, N. Koudas, D. Srivastava, V. J. Tsotras: Efficient Handling of Positional Predicates within XML Query Processing. Proc. XSym: 68–83 (2005)

Christian Mathis studied Computer Science from 1998 to 2004 at the University of Kaiserslautern. Since 2004 he is a Ph. D. Student at the DBIS research group lead by Prof. Härder. In the XTC project (XML Transaction Coordinator, http://www.xtc-project.de) he explores XML query processing.