Integrating Structural Joins into a Tuple-Based XPath Algebra

Christian Mathis
Databases and Information Systems
Department of Computer Science
University of Kaiserslautern, Germany
mathis@informatik.uni-kl.de

Abstract: Over the recent years, very little effort has been made to give XPath a proper algebraic treatment. The only 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.

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. Despite of the many algebra proposals regarding the standard XML query language *XQuery* [12, 17, 20], 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[1]. In this paper, we will extend the Natix Algebra (NAL) [3], which is—to our knowledge—the only algebra, specifically dealing with the compilation of XPath.

So, 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 eval-

[1] As you may convince yourself throughout this article.

uation as well as indexing techniques have been published, that qualify as *physical*[2] XML query operators. Among them, the most prominent representatives are the Structural Join (STJ) [1, 7, 14, 15], the Holistic Twig Join (HTJ) [5, 10], and the various path indexes like, for example, the D(k) index [6]. While being introduced in the context of *tree-based* algebras [12, 13], very little attempt has been made to integrate these concepts into a *tuple-based* XML algebra, such as NAL [17]. 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[3]. 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 operators STJ and HTJ. The extended algebra will be named NAL^{STJ}.

1.1 XML Algebras in the Literature

Although there is—to our knowledge—only one proposal explicitly dealing with the algebraic compilation and optimization of XPath queries [3], we give an overview over existing algebra approaches for XML queries in general and point out their XPath capabilities.

The TAX and TLC algebras [12, 13] 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, STJ and HTJ, 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) [17] takes a different approach, because it abstracts from trees as intermediate result structures. NAL operates on sequences of (homogeneous) tuples,

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

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

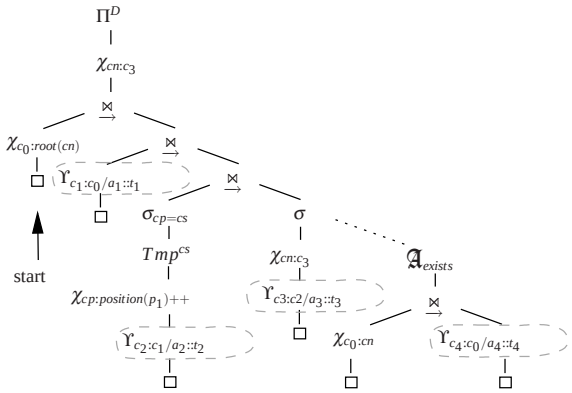

Figure 1: NAL Example

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 [8], these mappings keep track of the dynamic variable bindings during query processing. Reference [3] 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 [20]—called RSF[4] 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`[5]) 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 [8], the whole extent of XPath is covered. However, XPath-specific optimizations, as introduced in this article, have not been published in RSF so far. However, their integration would be possible.

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 $/a_1 :: t_1/a_2 :: t_2\ [position() = last()]/a_3 :: t_3\ [a_4 :: t_4]$ 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

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

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

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 sub-expression, is joined with t. In our example, the dependent expression is again a D-Join operator whose left sub-expression 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 sub-expression 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 $\chi_{cp:counter(p_2)++}$ 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} materializes 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 [23]) evaluation. The XPath-to-NAL translation process is described in [3] in great detail. Additionally, the authors provided some optimization techniques like stacked translation for outer paths, duplicate-elimination push down, and memoization[6]. In [4], certain algebraic equivalences were shown, which enable unnesting of queries with semi-correlated XPath predicates[7].

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, 5, 7] have pointed out that *set-at-a-time* processing of path steps provides better

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

[7] 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

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 [22] 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 [3] does not take important classes of physical operators into account, like the structural join and the holistic twig join[8]. 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 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* HTJ operator, followed by a subsequent selection. However, from the given representation, it is unclear how the mapping onto this HTJ 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 a *logical* 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.

By introducing structural joins, we can abstract from the explicit and implicit node-at-a-time path processing steps (e. g., the D-Join, and the selection due to a path predicate) inherent to the Natix algebra. This abstraction allows us to choose the adequate physical operators for the implementation of a logical plan. From the large set of possibilities for structural join implementation (i. e., stack based [1], hash based [15], index based [7], locking aware [14], etc.), we will gain more flexibility in the logical-to-physical mapping, and therefore extend the search space for cost-based query optimization. Surely, there will

[8] Although we recognize the hint towards that direction given in [18], we did not find any approach that properly introduces structural joins in NAL.

be situations node-at-a-time leads to a better performance than set-at-a-time. However, we think that this decision depends on physical properties and should, therefore, not be decided on a logical level.

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].

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 [22]. 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 rule set 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 [3]: 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), 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 the appendix. 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, and finally sequence-based merge ($\cup$) and intersect ($\cap$).

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 3:

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

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 its evaluation, an XML node identification mechanism (labeling scheme) is beneficial that can decide the relationship in question without a physical node access. All native XML database systems nowadays embody such a mechanism.

In case of all other shapes of the predicate p, we refer to the original definition of the selection operator in [3].

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. However, some care has to be taken for certain axes that may produce duplicates. Additionally, the question of output order arises[9]. For example, when using $\bowtie_p$ to evaluate the ancestor axis, the output may not reflect the document order (as required by XPath). Therefore, when using the structural join operator, we will keep these aspects in mind. The structural semi-join ($\ltimes_p$), the structural anti-join ($\rhd_p$), and the structural left-outer join ($\mathbb{X}\!\!\bowtie_p$) are defined accordingly.

Why do we claim this operator to be a *logical* operator? To answer this question, we first have to state that the distinction between logical and physical operators in XML algebras is

[9]Note, that the $\times$ operator on sequences, as defined in [3], returns an ordered result.

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, e. g., stack based [1], hash based [15], index based [6, 7], locking aware [14], etc. 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[10] c. It returns all nodes of a document in document order, complying 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.

Nest. In the following, we will not need the complex grouping capabilities of the general unary/binary grouping operator provided in NAL [17]. A simple nesting operator will do. Nesting is the complementary operator to unnesting. We assume the grouping operator in [17] 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 $\nu_{g:\bar{A}}(e) = \Gamma_{g;=A(e)\setminus A;id}(e)$.

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:

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

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:

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

[10] Note, in the following, we omit context-parameter c for simplicity

Sequence Merge and Intersect. The operators $\cup$ and $\cap$ are defined as the sequence-based, order preserving, and duplicate eliminating union and intersection on sequences of tuples having the same schema.

4 Introducing the Structural Join into a NAL Expression

In this section, we present a set of rewriting rules which substitute D-Join operators with structural joins. Each rule contains an operator pattern at the left-hand side. The corresponding right-hand side specifies how the operator tree has to be restructured. Note, a direct compilation from XPath to NAL^{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^{STJ} expression. Due to space restrictions, we cannot provide any reasoning about the correctness of the following rules. The necessary proofs can be found in the extended version of this paper [16]. After each rule application, the resulting operator tree can still be evaluated, because NAL^{STJ} is an extension of NAL. The introduction of structural joins is guided by the general rule:

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

At the left-hand side, the outer expression e_j generates a sequence of tuples containing an attribute c_j. For each tuple, this attribute is the starting point for the calculation of the axis step in the dependent unnest map expression. Φ is a function defined by a sequence of already translated algebra operators (i. e., Φ does not contain any D-Joins). Note, Φ may not only contain unary operators (as our notation suggests), but also binary ones (like $\ltimes$). However, because we assume Φ to be already translated, the rewriting depends on the single inner expression $\Upsilon_{c_i:c_j/a_i::t_i}$.

At the right hand side, 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. While, at the left-hand side, the evaluation contexts are neatly separated, at the right-hand side, they are intermixed. Therefore, this rewriting is only correct for certain Φ. We enumerate the variations of this rule for those Φ, for which the above rule would lead to an incorrect rewriting. In the following cases, Φ is split into three operators, of which two (Φ_1 and Φ_2) are again functions containing sub-expressions and the third is the operator of interest.

- $\Phi = \Phi_1 \circ Tmp^{cs} \circ \Phi_2$. This pattern leads to the following right-hand side, where the $Tmp^{cs}_{c_j}$ operator has the same semantics as in the stacked translation (see [3]):

  $$\Phi_1 \circ Tmp^{cs}_{c_j} \circ \Phi_2(e_j \ltimes_{c_i \theta_{a_i} c_j} \Upsilon_{c_i:\varphi_{t_i}}(\Box)) \tag{2}$$

  Due to the rewriting, the different evaluation contexts are not separated anymore. The operator has to detect groups of attributes belonging to the same context. In the rule, expression e_j binds attribute c_j, thus providing the outer context in which the

structural join is evaluated. $Tmp^{cs}_{c_j}$ detects groups based on c_j, i. e., whenever this attribute changes its value, the start of a new group is indicated. In the following, we will call operators that have been modified in this way *group aware*.

- $\Phi = \Phi_1 \circ \chi_{cp:counter(p)++} \circ \Phi_2$. For this pattern, we need to make the map operator group aware. Therefore, the expression $\chi_{cp:counter(p_i)++}$ has stacked-translation semantics (as defined in [3]):

$$\Phi_1 \circ \chi_{cp:counter(p_i)++} \circ \Phi_2(e_j \bowtie_{c_i \theta_{a_i} c_j} \Upsilon_{c_i:\varphi_{t_i}}(\Box)) \tag{3}$$

  Because the order matters for that pattern, we have to be careful to match XPath's semantics, which requires reverse document order, if a positional predicate is evaluated on a reverse axis. Therefore, if a_i is a reverse axis, we rewrite to:

$$\Phi_1 \circ \mathfrak{R}^G_{c_j} \circ \chi_{cp:counter(p_i)++} \circ \mathfrak{R}^G_{c_j} \circ \Phi_2(e_j \bowtie_{c_i \theta_{a_i} c_j} \Upsilon_{c_i:\varphi_{t_i}}(\Box)) \tag{4}$$

  As with the Tmp^{cs} operator, expression e_j provides the outer context, in which the structural join is evaluated. Therefore, the group reverse operator ($\mathfrak{R}^G_{c_j}$) groups by c_j. Likewise, the group-aware $counter()$ function resets its counter, when c_j changes. Note, we will abbreviate that function by $ct()$ in the following.

- $\Phi = \Phi_1 \circ \mathfrak{A}_{x:f} \circ \Phi_2$. If the pattern contains an aggregate function, we have to apply nesting first and evaluate the aggregate function on the nested attribute. Afterwards, the nested attribute can be projected out:

$$\Phi_1 \circ \Pi_{\overline{g}} \circ \mathfrak{A}_{x:f(\$g)} \circ \nu_{g:c_j} \circ \Phi_2(e_j \bowtie_{c_i \theta_{a_i} c_j} \Upsilon_{c_i:\varphi_{t_i}}(\Box)) \tag{5}$$

- $\Phi = \Phi_1 \circ Sort_{cn} \circ \Phi_2$. Here, a similar situation as in the previous rule can be found. We sort the nested group and unnest it again:

$$\Phi_1 \circ \mu_g \circ \mathfrak{S}^G_g \circ \nu_{g:c_j} \circ \Phi_2(e_j \bowtie_{c_i \theta_{a_i} c_j} \Upsilon_{c_i:\varphi_{t_i}}(\Box)) \tag{6}$$

For all other shapes of Φ, especially when Φ is the identity function, rule (1) can be applied. Also, when an operator has already been made *group aware*, as for example the Tmp^{cs} operator, (1) is used. If any Φ contains multiple matchings of the given pattern, they are applied in parallel. This typically happens for rules (2) and (3) in case of a positional predicate, i. e., $[position() = last()]$.

We conclude this section with the rewriting of a simplified version of the previous example: $/child :: a/child :: b\ [position() = last()]/child :: c$ (Figure 2). In the first step, e_1 and the depending sub-expression can be identified as depicted in Figure 2a. With Φ being the identity function, rule (1) can be applied. In Figure 2b, Φ contains a structural join, a selection, a Tmp^{cs}, and a map operator. Here, rules (2) and (3) are used "simultaneously". For Figure 2c, rule (1) applies again. Note, the position-handling operators have already been made group aware in the previous step.

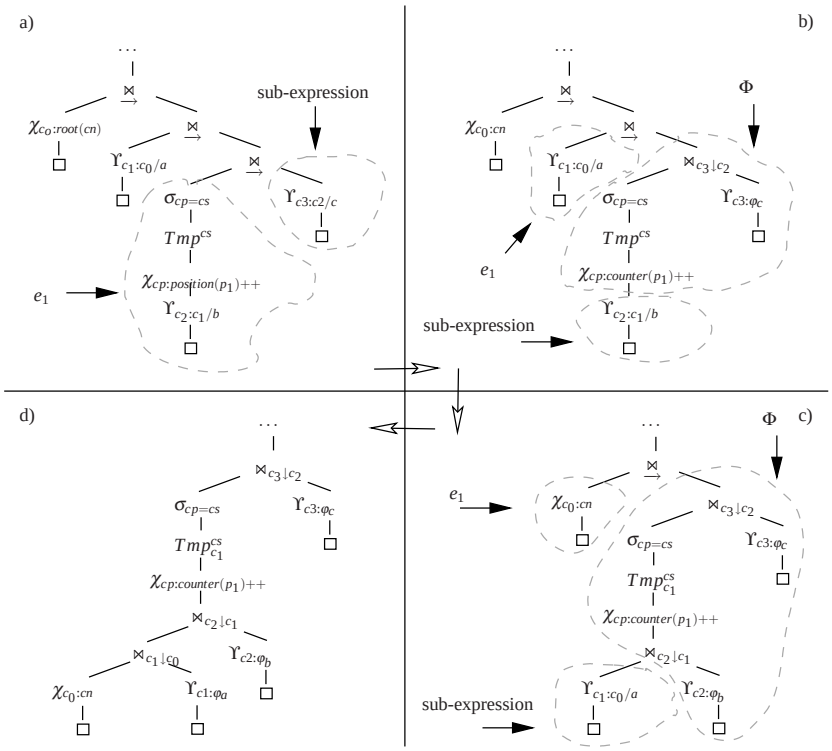

Figure 2: Translation Example

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[11]. Our query unnesting strategies are not covered by the rules in [4, 17]. Both contributions do not base their rewritings on the structural join operator.

The General Rules. By an examination of a typical operator tree, you can see that a pair of map operators ($\chi_{cn:c_i}$, $\chi_{c_0:cn}$) often "glues" an outer expression to the subscript of a

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

selection. Due to the XPath-to-NAL compilation, this is always the case when a predicate contains a path expression (for an example, see Figure 1). The inner map is the starting point for a cascade of operators, the first one of which is a structural join (in NAL^{STJ}). Our goal is to "extract" the inner path expression and join it with the outer expression. In some cases, we can abandon 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:

$$\sigma_{\Phi(\pi(\chi_{c_0:cn}(\Box)))}(\chi_{cn:c_0}(e_0)) = \Pi_{\overline{g}} \circ \sigma_{\Phi(\$g)} \circ \nu_{g:\overline{A(\pi')}}(e_0 \bowtie_{c_1 \theta c_0} \pi') \tag{7}$$

In 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 reestablished from the context attribute *cn*. Variable π is a NAL^{STJ} path expression depending on the context node given by the outer expression, i. e., $\pi = ((\chi_{c_0:cn}(\Box) \bowtie_{c_1 \theta_1 c_0} e_1) \ldots \bowtie_{c_n \theta_n c_0} e_n)$. Φ is—as in the previous rewriting rules—a sequence of NAL^{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 $\pi' = ((e_1 \bowtie_{c_2 \theta_1 c_1} e_2) \ldots \bowtie_{c_n \theta_n c_0} e_n)$. We denote this circumstance by the omission of the argument of π'. Furthermore, in the following, we will simply abbreviate $\pi(\chi_{c_0:cn}(\Box)))$ occurring in a selection subscript by π^{χ}. 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. While this rule is directly applicable, there are further refinements for special cases that provide for better performance results.

In contrast to all 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:

$$\begin{aligned} &\sigma_{\Phi(\pi_1^{\chi},\ldots,\pi_n^{\chi})}(\chi_{cn:c_0}(e_0)) \\ &= \Pi_{A(e_0)} \circ \sigma_{\Phi(\$g_1,\ldots,\$g_n)} \circ \nu_{g_1:\overline{A(\pi_1')}} \circ \cdots \circ \nu_{g_n:\overline{A(\pi_n')}}((e_0 \mathbin{⟕}_{c_1 \theta c_0} \pi_1') \cdots \mathbin{⟕}_{c_n \theta c_0} (\pi_n')) \end{aligned} \tag{8}$$

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*[12]. Another possible solution is to integrate the generation of nested groups into physical structural join operators, as sketched in [13].

[12] 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

By analyzing the most common cases in NAL^{STJ}, we identify Φ and provide specific unnesting redefinitions of the previous rules 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 sub-expressions 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 sub-expressions, 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:

$$\sigma_{e_2 \wedge e_3}(e_1) = \sigma_{e_2} \circ \sigma_{e_3}(e_1) = \sigma_{e_3} \circ \sigma_{e_2}(e_1) \tag{9}$$

Rewriting Disjunctive Predicates. Disjunctive predicates may be handled similarly to conjunctive ones using the sequence merge operator:

$$\sigma_{e_2 \vee e_3}(e_1) = \sigma_{e_2}(e_1) \cup \sigma_{e_3}(e_1) = \sigma_{e_3}(e_1) \cup \sigma_{e_2}(e_1) \tag{10}$$

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

$$\sigma_{\mathfrak{A}_{x:exists}(\pi^{\chi}) \vee e_2}(\chi_{cn:c_0}(e_0)) = \Pi^{D}_{A(e_0)} \circ \sigma_{(A(\pi') \neq \varepsilon) \vee e_2}(e_0 \mathbin{⟕}_{c_1 \theta c_0} \pi') \tag{11}$$

The notation $A(\pi') \neq \varepsilon$ essentially has the meaning $\forall a \in A(\pi') : a \neq \varepsilon$, i. e., π' has provided a join partner in the left outer join. In all other cases, when multiple path expressions in a general disjunction may occur, the query can be rewritten as:

$$\begin{aligned} &\sigma_{\Phi_1(\pi_1^{\chi}) \vee \Phi_2(\pi_2^{\chi})}(\chi_{cn:c_0}(e_0)) \\ &= \Pi_{A(e_0)} \circ \sigma_{(\Phi_1(\$g_1) \vee \Phi_2(\$g_2))} \circ \nu_{g_1:\overline{A(\pi_1')}} \circ \nu_{g_2:\overline{A(\pi_2')}}((e_0 \mathbin{⟕}_{c_1 \theta c_0} \pi_1') \mathbin{⟕}_{c_2 \theta c_0} \pi_2') \end{aligned} \tag{12}$$

In the following, every time a path expression participates in a disjunction, we use a left outer join operator instead of a full join. This guarantees, that we do not accidentally "throw away" intermediate results. For example, in the expression $a[b \vee c]$ we may not use an ordinary join between a and b, because then we would miss all a elements which should be part of the final result due to c.

Unnesting Existential Predicates. Sometimes plain path predicates like in $a[b/c]$ occur. In NAL^{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:

$$\sigma_{A_{x:exists}(\pi^{\chi})}(\chi_{cn:c_0}(e_0)) = e_0 \ltimes_{c_1 \theta c_0} \pi' \tag{13}$$

Note, on the right-hand side, π' is evaluated first, before the structural join is computed. Essentially this means, that π' is not evaluated in the context of e_1 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:

$$\sigma_{A_{x:exists}(\pi\chi)}(\chi_{cn:c_0}(e_0)) = \Pi^D_{A(e_0)}((e_0 \bowtie_{c_1\theta c_0} e_1)\cdots \bowtie_{c_n\theta c_{n-1}} e_n) \tag{14}$$

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

$$\sigma_{A_{x:\neg exists}(\pi\chi)}(\chi_{cn:c_0}(e_0)) = e_0 \rhd_{c_1\theta c_0} \pi' \tag{15}$$

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 the first rule above, 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^{STJ} expression[13] $\Pi^D(\chi_{cn:c_1}((\chi_{c_0:cn} \bowtie_{c_1\downarrow c_0} \Upsilon_{c_1:\varphi_a}) \ltimes_{c_2\downarrow c_1} (\sigma_{>3}(\Upsilon_{c_2:\varphi_b})))$. 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.

$$\sigma_{\mathfrak{A}_{x:exists}\circ\Phi_\theta(\pi_1^\chi,\pi_2^\chi))}(\chi_{cn:c_0}(e_0)) = \Pi^D_{A(e_0)} \circ \sigma_{(\$c_1\theta\$c_2)}((e_0 \bowtie_{c_1\theta c_0} \pi_1') \bowtie_{c_2\theta c_0} \pi_2') \tag{16}$$

In this rule Φ_θ is the compilation of the existential comparison as introduced in [3]. 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. This is also true for the duplicate elimination in rule (14).

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:

$$\sigma_{\Phi(\mathfrak{A}_{x:f}(\pi^\chi))}(\chi_{cn:c_0}(e_0)) = \Pi_{A(e_0)} \circ \sigma_{\Phi(\$x)} \circ \Gamma_{x;=A(e_0);f}(e_0 \bowtie_{c_1\theta c_0} \pi') \tag{17}$$

An Unnesting Example. We will conclude the discussion of query unnesting with an example. To save space, this example is presented using formulas. Consider the XPath expression $/desc :: a[child :: c = \text{“foo”} \vee count(desc :: b) > 3]$. The nested NAL^{STJ} query is:

$$\sigma_{(s_1)\vee(s_2)} \circ \chi_{cn:c_1}(e_0) \quad \text{where} \quad \begin{aligned} e_0 &= \chi_{c_0:root(cn)}(\Box) \bowtie_{c_1\Downarrow c_0} \Upsilon_{c_1:\varphi_a}(\Box) \\ s_1 &= \mathfrak{A}_{x:exists}((\chi_{c_0:cn}(\Box) \bowtie_{c_2\downarrow c_0} \Upsilon_{c_2:\varphi_c}(\Box)) \ltimes \text{“foo”}) \\ s_2 &= \mathfrak{A}_{x:count}(\chi_{c_0:cn}(\Box) \bowtie_{c_3\Downarrow c_0} \Upsilon_{c_3:\varphi_b}(\Box)) > 3 \end{aligned}$$

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

In the first step, we use Rule (11) to extract the *exists*() part of the disjunction:

$$\Pi_{c_0,c_1} \circ \sigma_{(c_2 \neq \varepsilon) \vee (s_2)} \circ \chi_{cn:c_1}(e_1) \quad \text{where} \quad e_1 = (e_0) \rtimilde_{c_2 \downarrow c_1} (\Upsilon_{c_2:\varphi_c}(\Box) \ltimes \text{“foo”}) \\ s_2 = \mathfrak{A}_{x:count()}(\chi_{c_0:cn}(\Box) \bowtie_{c_3 \Downarrow c_0} \Upsilon_{c_3:\varphi_b}(\Box)) > 3$$

In the second step, we use Rule (17) to extract the aggregate function from the disjunction:

$$\Pi_{c_0,c_1} \circ \sigma_{c_2 \neq \varepsilon \vee (\$x > 3)} \circ \Gamma_{x;=\{c_0,c_1,c_2\};count}(e_2) \quad \text{where} \quad e_2 = (e_1) \rtimilde_{c_3 \Downarrow c_0} \Upsilon_{c_3:\varphi_b}(\Box)$$

Finally, a slight optimization regarding expression e_1 can be pointed out: $\Upsilon_{c_2:\varphi_c} \ltimes \text{“foo”} = \Upsilon_{c_2:\varphi_c \wedge \text{“foo”}}$, i. e., the check for “foo” can simply be integrated into the node sequence access. In a physical algebra, this type of access could be supported by an index.

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, 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 HTJ join operator, a path index access, or onto the STJ 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 2d, 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 1 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

Table 1: 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)	$\mu_g(e_1) \bowtie_{c_2 \theta c_1} e_2 = \mu_g(e_1 \bowtie_{c_2 \theta c_1} e_2)$	$c_1 \notin A(g)$

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 1, we have found rules to push down a join over the special operators $Tmp^{cs}_{c_j}$ and $\chi_{cp:ct(p)++}$. However, their discussion is beyond the scope of this paper.

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 [11]). We refined the ORDPATH [19] 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 3). 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 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.

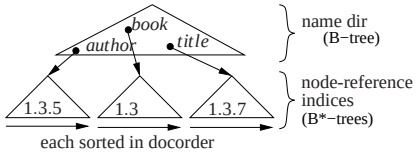

Figure 3: Element Index

Query Workload. The query workload depicted in Table 2 was run on four XMark [21] documents of size 120 KB, 1.2 MB, 12 MB, and 112 MB (factors 0.001, 0.01, 0.1, 1). To compare the raw performance of the given strategies, we switched off isolation mechanisms in XTC, thus, no locking overhead occurs. 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

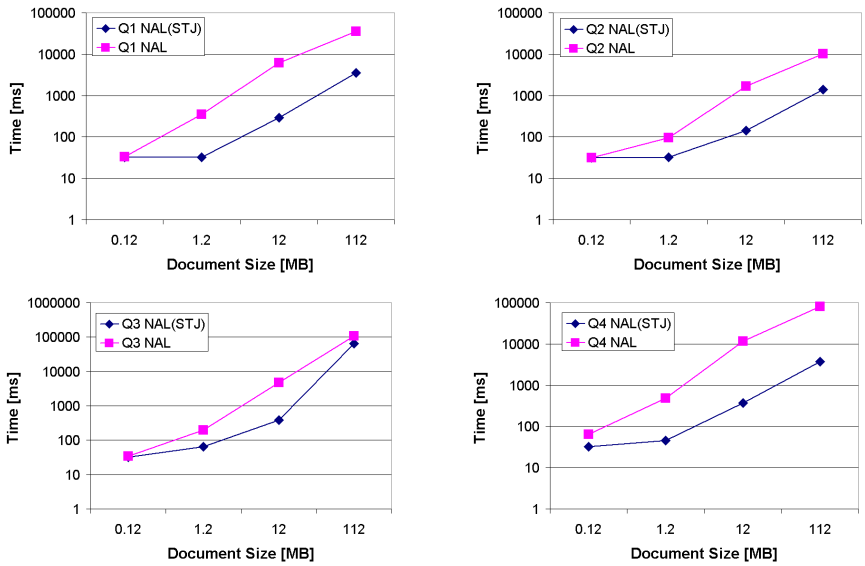

Figure 4: Queries Q1, Q2, Q3, and Q4

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

Table 2: 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

faster (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^{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^{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 [24]. 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 free an operator plan from implicit (selections) and explicit (D-Join) node-at-a-time processing steps. Note, this is accomplished at the logical level only; a physical implementation may freely choose to implement a structural join in a node-at-a-time manner [14], nevertheless. Even hash-based strategies may be applied [15]. But the decision to do so depends on physical issues and should not be determined at the logical level.

Finally, 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. And more sophisticated mappings are still to come, from which we hope to gain further improvements.

Acknowledgements. I would like to thank Theo Härder, Jose de Aguiar Moraes Filho 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 et al.: Structural Joins: A Primitive for Efficient XML Query Pattern Matching. Proc. ICDE: 141–152 (2002)

[2] 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)

[3] M. Brantner, C.-C. Kanne, S. Helmer, G.Moerkotte: Full-fledged Algebraic XPath Processing in Natix. Proc. ICDE: 705–716 (2005)

[4] M Brantner, C.-C. Kanne, S. Helmer, G. Moerkotte: Algebraic Optimization of Nested XPath Expressions. Proc. ICDE: 128 (2006)

[5] N. Bruno, N. Koudas, D. Srivastava: Holistic twig joins: Optimal XML pattern matching. Proc. SIGMOD: 310–321 (2002)

[6] Q. Chen, A. Lim, K. W. Ong: D(k)-Index: An Adaptive Structural Summary for Graph-Structured Data. Proc. SIGMOD: 134–144 (2003)

[7] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, C. Zaniolo: Efficient Structural Joins on Indexed XML Documents. Proc. VLDB: 263–274 (2002)

[8] D. Draper, et. al.: XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Specification. http://www.w3.org/TR/xquery-semantics/

[9] M. F. Fernandez, J. Hidders, P. Michiels, J. Simeon, R. Vercammen: Optimizing Sorting and Duplicate Elimination. Proc DEXA: 554-563 (2005)

[10] M. Fontoura, V. Josifovski, E. Shekita, B. Yang: Optimizing Cursor Movement in Holistic Twig Joins. Proc. 14th CIKM: 784–791 (2005)

[11] T. Härder, M. Haustein, C. Mathis, M. Wagner: Node Labeling Schemes for Dynamic XML Documents Reconsidered. Data & Knowledge Engineering 60: 126–149 (2007)

[12] H. Jagadish and L. Lakshmanan and D. Srivastava and K. Thompson: TAX: A Tree Algebra for XML. Proc. DBPL: 149–164 (2001)

[13] S. Paparizos, Y. Wu, L. V. S. Lakshmanan, H. V. Jagadish: Tree Logical Classes for Efficient Evaluation of XQuery. Proc. SIGMOD: 71–82 (2004)

[14] Mathis, Ch., Härder, T., Haustein, M.: Locking-Aware Structural Join Operators for XML Query Processing. Proc. SIGMOD: 467–478 (2006)

[15] Mathis, Ch., Härder, T.: Hash-Based Structural Join Algorithms, Proc. DATAX'06, LNCS 4254, Springer-Verlag, 136–149 (2006)

[16] Mathis, Ch.: Extending a Tuple-Based XPath Algebra to Enhance Evaluation Flexibility. Internal Report. http://wwwdvs.informatik.uni-kl.de/pubs/papers/M07.Internal.html

[17] N. May, S. Helmer, G. Moerkotte: Nested Queries and Quantifiers in an Ordered Context. Proc. ICDE: 239–250 (2004)

[18] N. May, M. Brantner, A. Böhm, C.-C. Kanne, G. Moerkotte: Index vs. Navigation in XPath Evaluation. Proc. XSym: 16–30 (2006)

[19] 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)

[20] C. Re, J. Siméon, M. Fernández: A Complete and Efficient Algebraic Compiler for XQuery. Proc. ICDE: 14 (2006)

[21] A. Schmidt, et. al: XMark: A Benchmark for XML Data Management. Proc. VLDB: 974–985 (2002)

[22] Y. Wu, J. M. Patel, H. V. Jagadish: Structural Join Order Selection for XML Query Optimization. Proc. ICDE: 443–454 (2003)

[23] W3C Recommendation: XML Path Language (XPath), Version 1.0 (1999). http://www.w3.org/TR/xpath

[24] V. Zografoula, et. al: Efficient Handling of Positional Predicates within XML Query Processing. Proc. XSym: 68–83 (2005)

A NAL Overview

Table 3: Relevant NAL Operators taken from [3]

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	$e_1 \times e_2 := (\alpha(e_1))\overline{\times}e_2) \oplus (\tau(e_1)\overline{\times}e_2)$
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	$e_1 \ltimes_p e_2 := \begin{cases} \alpha(e) \oplus (\tau(e_1) \ltimes_p e_2) & : \quad \exists x \in e_2 : p(\alpha(e_1) \circ x) \\ \tau(e_1) \ltimes_p e_2 & : \quad else \end{cases}$
Anti-Join	$e_1 \rhd_p e_2 := \begin{cases} \alpha(e) \oplus (\tau(e_1) \rhd_p e_2) & : \quad \nexists x \in e_2 : p(\alpha(e_1) \circ x) \\ \tau(e_1) \rhd_p e_2 & : \quad else \end{cases}$
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	$Sort_a(e) := Sort_a(\sigma_{a<\alpha(e).a}(\tau(e))) \oplus \alpha(e) \oplus Sort_a(\sigma_{a\geq\alpha(e).a}(\tau(e)))$
Singleton Scan	$\Box := [\{\}]$