Put a Tree Pattern in Your Algebra

Philippe Michiels
University of Antwerp

George A. Mihăilă
IBM Research

Jérôme Siméon
IBM Research

Abstract

To address the needs of data intensive XML applications, a large number of efficient tree pattern algorithms have been proposed. Still, most XQuery compilers do not support those algorithms. This is due in part to the lack of support for tree patterns in most algebras, but also because identifying which part of a query plan should be evaluated as a tree pattern is a hard problem. In this paper, we extend a tuple algebra for XQuery with a tree pattern operator, and present rewritings suitable to introduce such operators in query plans. We demonstrate the robustness of the proposed rewritings under syntactic variations commonly found in real queries. The proposed tree pattern operator can be implemented using popular algorithms such as Twig joins and Staircase joins. We report on experiments which suggest heuristics useful to decide which algorithm should be used for a given plan.

1 Introduction

Efficient evaluation of path expressions is crucial for the overall performance of any XQuery engine. For that reason, the development of algorithms based on the notion of tree-pattern has been a key focus of research on XML query processing [4, 15, 23, 19, 9, 18, 16]. Most XQuery algebras [11, 24, 27, 14, 1] do not support tree patterns, focusing instead on recovering important relational optimizations [11, 24, 27, 14, 1] and on how to support larger fragments of the language [3, 27]. In those approaches, path expressions are typically compiled into nested maps with navigational primitives, missing opportunities for using tree pattern algorithms. The TAX algebra [6, 26] used in the Timber system is the only one to include support for tree patterns. However, the approach is still limited to a fragment of XQuery and the absence of support for tuple operators makes it more difficult to recover important relational optimizations. In this paper, we answer three important questions: how can tree pattern processing be integrated in a tuple algebra for XQuery, how can the compiler decide when a fragment of the query plan can be evaluated using a tree pattern, and how to decide which tree pattern algorithm should be used.

There are several reasons why detecting tree patterns in arbitrary XQuery plans is difficult. Very simple path expressions, such as query **Q1a** on Figure 1, might already be in the form of a tree pattern. However, it is often the case that such a tree pattern is written as a combination of FLWOR and path expressions, as in queries **Q1b** and **Q1c** on Figure 1, which are both equivalent to **Q1a**. Secondly, tree patterns correspond only to very specific fragments of XPath, notably without complex predicates or backward axis, while most queries usually feature complex combinations of XPath primitives which must be broken up into several tree patterns. For instance, query **Q2** on Figure 1 should be split into two tree patterns connected by a selection predicate on the name, while both query **Q3** and **Q4** require a more complex treatment in order to properly compute the position predicate. Finally, subtle aspects of the semantics of path expressions must be taken into account. For instance, despite being almost identical to **Q1b**, query **Q5** is not a tree pattern, since it may not return the names in document order, and must be split into two tree patterns composed through a map operator.

In this paper, we present an algebraic framework and compilation techniques that allow a complete XQuery compiler to detect when tree pattern algorithms can be used. Instead of relying on syntax to identify tree patterns, we use an approach based on the semantics of the query, along with a two phase rewritings process which provides a robust way to identify which part of a query can be evaluated as a tree pattern. We believe this work bridges an important gap between the literature on tree pattern algorithms and the literature on algebraic XQuery optimization. We implemented the proposed approach in the Galax XQuery 1.0 processor [14][1] and experiments show the robustness of the approach for complex tree pattern queries.

The main contributions of the paper are as follows:

- We extend the tuple algebra of [27] with a tree pattern operator. That operator is designed to integrate with other tuple operations and can be implemented using popular algorithms such as Twig joins [4, 15] and Staircase joins [18].

[1] The corresponding implementation can be tried by downloading Galax version `0.6.5` or later, at `http://www.galaxquery.org/`.

Q1a `$d//person[emailaddress]/name`

Q1b
```
(for $x in
   $d//person[emailaddress]
 return $x)/name
```

Q1c
```
let $x :=
  for $y in $d//person
  where $y/emailaddress
  return $y
return $x/name
```

Single Tree Patterns

Q2 `$d//person[name = "John"]/emailaddress`

Q3 `$d//person[1]/name`

Q4 `$d//person[name = "John"]/emailaddress[1]`

Q5
```
for $x in
 $d//person[emailaddress]
return $x/name
```

Multiple Tree Patterns

Figure 1. Tree Patterns Queries

- We present rewritings that normalize queries with tree patterns to prepare for their detection at the algebraic level. The rewritings are sound in the face of XPath's complex semantics and are robust for syntactic variations most commonly found in queries.
- We present algebraic optimizations rules to introduce tree-pattern operators in query plans whenever possible.
- We present experiments showing the robustness of the proposed compilation approach in practice.
- We use the resulting compiler to compare the performance of well-known tree pattern algorithms in the context of query plans for the XMark benchmark. This comparison suggests useful heuristics for deciding which algorithm should be used for a given plan.

The rest of the paper is organized as follows. Section 2 gives an overview of our compilation approach and illustrates how it addresses the issues about tree pattern detection raised earlier in this section. Section 3 describes the rewritings used to normalize tree patterns in arbitrary queries. Section 4 presents the algebraic tree pattern operator and the corresponding optimizations. Section 6 contains the experimental evaluation and discusses heuristics to decide which algorithm to use. Section 7 reviews the related work, and Section 8 concludes the paper.

2 Tree Pattern Compilation

The compilation proceeds through several phases which are shown on Figure 2.

XQuery Normalization. The first phase consists in normalizing the query into the XQuery Core, as specified in [12, 27]. The normalization rules for path expressions (corresponding to the following grammar productions (68), (69), (70), (71), and (81) in [2]), are described in detail

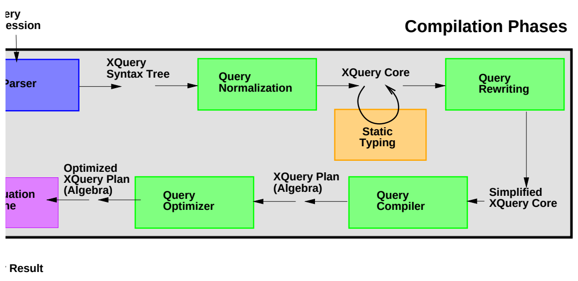

Figure 2. Compilation Architecture

in [12]. For instance, normalization applied to the simple path **(Q1a)** on Figure 1 results in the following XQuery Core expression[2].

Q1a-n
```
ddo(
 let $seq :=
  ddo(
   let $seq := ddo($d),
   let $last := fn:count($seq)
   for $dot at $position in $seq
   return
    let $seq := ddo(descendant::person),
    let $last := fn:count($seq)
    for $dot at $position in $seq
    where
     typeswitch (child::emailaddress)
      case $v2 as numeric() return
       $position = $v2
      default $v3 return
       fn:boolean($v3)
    return $dot),
 let $last := fn:count($seq)
 for $dot at $position in $seq
 return child::name)
```

[2]To simplify exposition, we assume here that the step `$d//person` is normalized similarly to `$d/descendant::person`.

Lines 4-7 and 1-2,18-20 correspond to the / expression, lines 3 and 8-17 correspond to the predicate, and lines 8, 12 and 20 correspond to the steps `descendant::person`, `child::emailaddress` and `child::name` respectively. One benefit of this approach is that it deals with all of XPath's semantics, including sorting by document order and duplicate elimination (through calls to the special function `ddo` on lines 1, 2, 4, and 8), the binding of context information such as `position()` (on lines 6, 10, and 19) and `last()` (on lines 5, 9, and 18), and the case where predicates are applied to numeric values (the first branch of the `typeswitch` expression on lines 13-14).

Normalization is crucial in our context as it exposes the implicit iteration in XPath's E_1 / E_2 and $E_1 [E_2]$ operators. This is the first step toward providing a uniform treatment for combinations of FLWOR and path expressions.

XQuery Core Rewriting. It is important to note that normalization is applied mechanically to each XQuery expression. As a result, different queries (e.g., **Q1b** and **Q1c** in Figure 1) result in different normalized queries. The purpose of the next compilation phase is to rewrite expressions corresponding to tree patterns to remove non-relevant syntactic differences. The rewriting phase uses a set of equivalences expressed over the XQuery core, and prepares for algebraic compilation. The corresponding rewritings are presented in details in Section 3. After rewritings, queries corresponding to tree patterns are always in the same form, which is a specific combination of step expressions, iteration, and calls to sorting by document order and duplicate elimination. For example, the following query is the rewritten form of all the three queries **Q1a**, **Q1b**, and **Q1c**.

```
ddo(                                   Q1-tp
 for $dot in ddo(
   for $dot in ddo(
    for $dot in $d
    return descendant::person)
   where child::emailaddress
   return $dot)
 return child::name)
```

Algebraic Compilation. The resulting expression is then compiled into the algebraic plan **P1**. That phase uses the algebra and compilation rules defined in [27].

```
ddo(MapToItem                                    P1
 {TreeJoin[child::name](IN#dot)}
 (MapFromItem{[dot : IN]}
  (ddo(MapToItem{IN#dot}
   (Select
    {fn:boolean(
      TreeJoin[child::emailaddress](IN#dot))}
    (MapFromItem{[dot : IN]}
     (ddo(MapToItem
      {TreeJoin[descendant::person](IN#dot)}
      (MapFromItem{[dot : IN]}($d))))))))))
```

For conciseness, the query plans are written in the functional notation used in [27], where each operator has a name (e.g., Select), a set of inputs sub-plans given in parenthesis, and possibly some dependant sub-plans written in curly braces. The evaluation of a dependant sub-plan depends on each tuple or item (denoted by IN) returned by the input sub-plans. The original plans compiled for a tree pattern combines map operators (MapFromItem and MapToItem) to perform iteration, navigational primitives (TreeJoin), and calls to special functions (such as `ddo`). Other operators include `[dot : Op]` which constructs a new tuple with a field `dot`, `IN#dot` which accesses the `dot` field in the input tuple, and Select which stands for relational selection.

Algebraic Optimization. Once in algebraic form, the optimization phase includes special-purpose optimization rules to introduce a tree pattern operator (written TupleTreePattern in our algebra). The corresponding optimization rules are presented in Section 4. In the case of query **Q1a**, **Q1b**, and **Q1c**, the resulting plan looks very simple, with a single TupleTreePattern corresponding to the expected tree pattern, as follows.

```
MapToItem{IN#dot}
 (TupleTreePattern
  [IN#dot/descendant::person
   [child::emailaddress]/child::name{dot}]
  (MapFromItem{[dot : IN]}(IN#d)))
```

This TupleTreePattern accesses the `dot` field from the input tuple, evaluates the tree pattern to produces a sequence of output tuples with a new output `dot` field corresponding to the names of the persons matching that tree pattern. For instance, the following plan corresponds to query **Q2**, and combines two tree patterns with a selection predicate.

```
MapToItem{IN#dot}
 (TupleTreePattern
  [IN#dot/child::emailaddress{dot}]
  (Select{TreeJoin[child::name](IN#dot)="John"}
   (TupleTreePattern
    [IN#dot/descendant::person{dot}]
    (MapFromItem{[dot : IN]}($d)))))
```

The optimization rules applied during that phase verify two important properties. First, they are always directed in a way that creates bigger tree patterns, which means they detect the largest set of consecutive algebraic operators that can be combined into a single tree pattern. Second, each rule works specifically with combinations of algebraic operators which have a tree pattern semantics, which ensures that intermediate operators (e.g., as for the Select operator in **Q2**) are preserved in the final plan.

Choosing a tree pattern algorithm. Finally, the last compilation phase decides which tree pattern algorithm to use (e.g., TwigJoin, Staircase join, nested-loop). We will see in Section 6 that deciding which algorithm to use is non trivial, and we will suggest some heuristics which work in simple cases, depending on the shape of the tree pattern and their location in the query plan.

3 Tree Pattern Rewritings

In this section, we focus on the rewriting techniques needed to prepare XQuery expressions for tree pattern detection. We write the corresponding transformations using inference rule notations similar to those used in [12]. To illustrate the effect of each set of rewritings, we apply them on the normalized expression **Q1a-n** given at the beginning of Section 2. Due to space limitations, we focus only on the most important rewritings. The full set of rewritings notably includes variable renaming (which is necessary to handle **Q1b**, and **Q1c**), and can be found in [25]. The full set of rewritings is provably sound and complete for a non-trivial fragment of XQuery (also given in [25]). However, a formal study of the properties of those rewritings is beyond the scope of this paper. The experimental evaluation in Section 6 confirms the robustness of those rewritings in practice.

Type rewritings. The first rules deal with `typeswitch` expressions resulting from the compilation of XPath predicates. The first rule removes case clauses which are sure to be unused, while the second rule bypasses the typeswitch in case one clause is sure to be used. Those rules rely on type information, which can be obtained using XQuery's static typing feature. We refer the reader to previous work on XQuery static typing [12, 10] for more details.

$$\frac{\begin{array}{c}\texttt{typeswitch (}\mathit{Expr}_0\texttt{)}\\ \texttt{case } \mathit{Type}_1 \texttt{ as \$v1 return } \mathit{Expr}_1\, \mathit{CaseClauses}\\ \mathit{statEnv} \vdash \mathit{Expr}_0 : \mathit{Type}_0\\ \mathit{statEnv} \vdash \mathit{Type}_0 \cap \mathit{Type}_1 = \emptyset\end{array}}{\texttt{typeswitch (}\mathit{Expr}_0\texttt{)}\ \mathit{CaseClauses}}$$

$$\frac{\begin{array}{c}\texttt{typeswitch (}\mathit{Expr}_0\texttt{)}\\ \texttt{case } \mathit{Type}_1 \texttt{ as \$v1 return } \mathit{Expr}_1\, \mathit{CaseClauses}\\ \mathit{statEnv} \vdash \mathit{Expr}_0 : \mathit{Type}_0\\ \mathit{statEnv} \vdash \mathit{Type}_0 \subset \mathit{Type}_1\end{array}}{\texttt{let \$v1 := } \mathit{Expr}_0 \texttt{ return } \mathit{Expr}_1}$$

Applying the previous rewriting rules to lines 12-16 on the normalized expression for Q1 results in the following.

```
...
11.     where
12.       let $v3 := child::emailaddress
13.       return fn:boolean($v3)
14.     return $dot),
...
```

FLWOR rewritings. We then apply a set of rewritings to remove unnecessary `let` and `for` clauses. The first two rules remove unused let bindings and perform variable inlining. The third rule removes unused index variables in `for` clauses. Those rules rely on an auxiliary judgment that computes the usage count of a variable in a given expression.

$$\frac{\begin{array}{c}\texttt{let \$x := } \mathit{Expr}_1 \texttt{ return } \mathit{Expr}_2\\ \texttt{\$x}\ \text{usage in } \mathit{Expr}_2 = 0\end{array}}{\mathit{Expr}_2}$$

$$\frac{\begin{array}{c}\texttt{let \$x := } \mathit{Expr}_1 \texttt{ return } \mathit{Expr}_2\\ \texttt{\$x}\ \text{usage in } \mathit{Expr}_2 = 1\end{array}}{[\mathit{Expr}_2 | \$x \Rightarrow \mathit{Expr}_1]}$$

$$\frac{\begin{array}{c}\texttt{for \$x at \$i in } \mathit{Expr}_1 \texttt{ return } \mathit{Expr}_2\\ \texttt{\$i}\ \text{usage in } \mathit{Expr}_2 = 0\end{array}}{\texttt{for \$x in } \mathit{Expr}_1 \texttt{ return } \mathit{Expr}_2}$$

Applying these rules results in the following.

```
ddo(
 for $dot in ddo(
  for $dot in ddo($d)
  return
   for $dot in ddo(descendant::person)
   where fn:boolean(child::emailaddress)
   return $dot )
 return child::name)
```

Loop fusion. The next transformation is a loop-fusion rewritings that is necessary to impose the proper nesting on the evaluation of predicates.

$$\frac{\begin{array}{l}\texttt{for \$x in } \mathit{Expr}_1\, \texttt{(where } \mathit{Cond}_1\texttt{)? return}\\ \quad \texttt{for \$y in } \mathit{Expr}_2\, \texttt{(where } \mathit{Cond}_2\texttt{)? return } \mathit{Expr}_3\end{array}}{\begin{array}{l}\texttt{for \$y in}\\ \quad \texttt{for \$x in } \mathit{Expr}_1\, \texttt{(where } \mathit{Cond}_1\texttt{)? return } \mathit{Expr}_2\\ \texttt{(where } \mathit{Cond}_2\texttt{)? return } \mathit{Expr}_3\end{array}}$$

Applying loop fusion to our example results in the following expression.

```
ddo(
 for $dot in ddo(
  for $dot in
   for $dot in ddo($d)
   return ddo(descendant::person)
  where fn:boolean(child::emailaddress)
  return $dot)
 return child::name)
```

It is important to note that this rule does not hold in the case one of the `for` expressions contains an index variable, as would be the case if one of the expression uses a the context position. For instance, consider the result of applying all the previous rewriting rules to the query `$d//person[position()=1]`.

```
ddo(for $dot in
  ddo(for $dot in $d
         return descendant-or-self::node())
 return
  for $dot at $pos in ddo(child::person)
  where $pos = 1 return $dot)
```

Applying loop fusion in this case would result in the context position being computed with respect to the set of all persons in the document, rather than once for each individual set of persons that are children of a given node in the document.

Document order rewritings. Finally, we apply a set of rewritings whose purpose is to normalize the use of `ddo` calls. The first two rewritings remove unnecessary `ddo` calls, while the last rewriting makes sure intermediate expressions are wrapped into `ddo` calls.

$$\frac{\texttt{ddo(}\mathit{StepExpr}\texttt{)}}{\mathit{StepExpr}}$$

$$\frac{\begin{array}{c}\texttt{ddo(}\mathit{Expr}\texttt{)}\\ \mathit{statEnv} \vdash \mathit{Expr} : \mathit{Type}\\ \mathit{Type} \subset \texttt{node}()\end{array}}{\mathit{Expr}}$$

$$\frac{\begin{array}{c}\texttt{ddo(for \$x in } \mathit{Expr}_1\\ \quad\texttt{where } \mathit{Cond}_2 \texttt{ return } \mathit{Expr}_2\texttt{)}\\ \mathit{Expr}_1 \text{ order independant}\\ \mathit{Expr}_2 \text{ order independant}\end{array}}{\begin{array}{l}\texttt{ddo(for \$x in ddo(}\mathit{Expr}_1\texttt{)}\\ \quad\texttt{where } \mathit{Cond}_2 \texttt{ return ddo(}\mathit{Expr}_2\texttt{))}\end{array}}$$

Applying these rules results in the expression **Q1-tp** given in Section 2.

4 Algebraic Tree Pattern Optimization

In this section, we describe the tree pattern operator, and the algebraic treatment of XQuery path expressions.

4.1 TupleTreePattern Operator

We extend the algebra of [27] with a new operator for tree pattern evaluation: TupleTreePattern[TP](Op), where TP is the tree pattern being applied, and Op is the algebraic plan computing the input for the operator. Tree patterns are expressed using a small fragment of XPath which is described by the following grammar, with $Axis$ and $NodeTest$ being defined as in XPath.

$$\begin{array}{lcl}
\mathit{TreePattern} & ::= & \texttt{IN}\#\mathit{FieldName}(/\mathit{Pattern})?\\
\mathit{Pattern} & ::= & \mathit{Step}([\mathit{Pattern}])*(/\mathit{Pattern})?\\
\mathit{Step} & ::= & \mathit{Axis}\ \mathit{NodeTest}\{\mathit{FieldName}\}?\\
\mathit{FieldName} & ::= & \texttt{QName}
\end{array}$$

The TupleTreePattern operator takes a sequence of tuples as input and produces a sequence of tuples as output. The input field containing the context nodes being processed is given at the beginning of the pattern. The tree pattern used as a parameter for the operator includes annotations for the nodes that must be returned and the corresponding fields in the output tuples. The signature for that operator is given below. `S(...)` denotes an (ordered) sequence, `S(`τ`)` denotes a sequence of tuples of type τ, `S(`i`)` denotes a sequence of items in the XQuery Data Model (XDM) [28], $\mathit{TreePattern}\{\mathtt{q}_1,...,\mathtt{q}_n\}$ denotes a tree pattern containing the output fields $\mathtt{q}_1,...,\mathtt{q}_n$.

$$\begin{array}{c}\text{TupleTreePattern}[\mathit{TreePattern}\{\mathtt{q}_1,...,\mathtt{q}_n\}](\mathtt{S}(\tau))\\ \longrightarrow\\ \mathtt{S}([\mathtt{q}_1{:}\mathtt{S}(i);...;\mathtt{q}_n{:}\mathtt{S}(i)])\end{array}$$

A TupleTreePattern returns all bindings matching the tree pattern, in a root-to-leaf lexical order, which is consistent with the semantics of TwigJoins [4]. Those bindings are returned as fields within the output tuples, based on the field annotations in the pattern. This semantics is illustrated on the following example.

```
TreePattern
 [IN#x/descendant::a/child::c{y}[@id]/child::d{z}](
  [ x : <a><c id="1"><d id="2"/><d id="3"/></a> ],
  [ x : <a><c/><a/> ],
  [ x : <a><c id="4"><d id="5"/></c><c id="6"/></a> ]
)
= ([ x : <a><c id="1"><d id="2"/><d id="3"/></c></a>;
     y : <c id="1"><d id="2"/><d id="3"/></c>;
     z : <d id="2"/> ],
   [ x : <a><c id="1"><d id="2"/><d id="3"/></c></a>;
     y : <c id="1"><d id="2"/><d id="3"/></c>;
     z : <d id="3"/> ],
   [ x : <a><c id="4"><d id="5"/></c><c id="6"/></a>;
     y : <c id="4"><d id="5"/></c>;
     z : <d id="5"/> ])
```

Note that the TupleTreePattern operator essentially behaves as a dependant join. In the above example, the second tuple for which there is no match does not appear in the result, the first tuple which matches the pattern twice results in two output tuples, and the last tuple results only in one tuple for the `c` element that contains a `d` child.

Definition 4.1:[Extraction Point of an XPath expression] The extraction point of a path expression is the last step of the path that is not part of a predicate. ■

Note that the TreePattern operator may result in bindings in which some of the fields have duplicates and that some of those bindings may not be in document order.

However, the semantics coincide with the XPath semantics in the case there is only an output field on the extraction point of the tree pattern.

4.2 TreePattern optimization

We now consider the path expression **Q1a** presented in Section 2. Applying the normalization and rewriting phases described in the previous section results in the expression **Q1-tp** which is then compiled in the algebra by applying the compilation rules in [27], resulting in the plan **P1**. This plan features map operators, TreeJoin, and `fs:ddo`. In the rest of the section, we illustrate the algebraic optimizations that enable the detection of a single tree pattern operator for that plan.

Figure 3 summarizes the algebraic rewritings necessary to detect TupleTreePattern in query plans. We focus on the most important rewritings. Due to space limitations, some additional "clean-up" rewritings used to make the detection more robust in complex plans are not discussed here. For instance, the last `fs:ddo` operation is removed since the TreeJoin is always applied to one item.

From TreeJoin to TupleTreePattern

The first step in picking up tree patterns from the algebraic query plan is to rewrite TreeJoin operators, which operate on items, into TupleTreePattern operators which operate on tuples. This step is handled by the rewrite rules (a) and (b), each applying to TreeJoin occurences in very specific contexts. Rule (a) is the most general and can always be applied to replace an occurence of the TreeJoin operator and introduces and extra MapToItem that converts the output from tuples to items, to emulate the output of the TreeJoin. Rule (b) operates on a more specific case, in case a MapToItem operator is already present as it is often the case, notably when the step expression resulting from normalization is within a `for` clause. This rule is applied before rule (a). Note that rule (b) requires the presence of a distinct-document-order operation, which is necessary since the resulting TupleTreePattern returns nodes in document order. Applying rules (a) and (b) to the TreeJoin on lines 2, 7 and 10 in **P1**, results in the plan **P2** below.

```
1.  fs:ddo(MapToItem{IN#out}                                P2
2.   (TupleTreePattern[IN#dot/child::name{out}]
3.     (MapFromItem{[dot : IN]}
4.       (fs:ddo(MapToItem{IN#dot}
5.         (Select{fn:boolean(MapToItem{IN#out}
6.           (TupleTreePattern
7.               [IN#dot/child::emailaddress{out}]
8.             (IN)))}
9.           (MapFromItem{[dot : IN]}
10.            (fs:ddo(MapToItem{IN#out}
11.              (TupleTreePattern
12.                  [IN#dot/descendant::person{out}]
13.                (MapFromItem{[dot : IN]}($d)))))))))))
```

Eliminating Item-Tuple Conversions

Note that this rewriting may result in slightly more complex plans notably because of the introduction of extra maps, and that the TupleTreePatterns are still applied in a nested loop fashion. The following rule, (c) introduces bulk TreePatterns by collapsing MapFromItem operators applied to MapToItem operators — which in turn have a TupleTreePattern as independent subexpression. Before we can collapse these item-tuple conversions, we need to eliminate the fs:ddo call, which can be done with rule (f). Applying rules (f) and (c) to on lines 9-10 in **P2** above, results in the plan **P3** below.

```
1.  fs:ddo(MapToItem{IN#out}                                P3
2.   (TupleTreePattern[IN#dot/child::name{out}]
3.     (MapFromItem{[dot : IN]}
4.       (fs:ddo(MapToItem{IN#dot}
5.         (Select{fn:boolean(MapToItem{IN#out}
6.           (TupleTreePattern
7.               [IN#dot/child::emailaddress{out}]
8.             (IN)))}
9.           (TupleTreePattern
10.              [IN#dot/descendant::person{dot}]
11.            (MapFromItem{[dot : IN]}($d)))))))))
```

Merging individual tree patterns

The final set of rules are used to merge compositions of single-step tree patterns into multi-step tree patterns occur next to each other. Rule (d) deals with sequences of consecutive steps, while rule (e) deals with predicate branches in the pattern. In the plan **P3** above, we first have to apply (e) on lines 5-10, resulting plan **P4** below, in which the Select operation to be removed and a predicate branch added to the tree pattern. Note that the `fs:ddo` in **P3** can be removed with rule (f), since it operates over a TupleTreePattern, which includes `fs:ddo` semantics.

```
1.  fs:ddo(MapToItem{IN#out}                                P4
2.   (TupleTreePattern[IN#dot/child::name{out}]
3.     (MapFromItem{[dot : IN]}
4.       (MapToItem{IN#dot}
5.   (TupleTreePattern
6.       [IN#dot/descendant::person{dot}
7.              [child::emailaddress]]
8.       (MapFromItem{[dot : IN]}($d))))))
```

Finally, applying rule (d) again on lines 3-4, followed by rule (e) on the TreePatterns on lines 2 and 5-7, result in the final expected plan **P5** below, where the tree pattern has been fully recognized by the compiler.

```
1.  MapToItem{IN#out}                                       P5
2.   (TupleTreePattern
```

Replacing TreeJoins with TupleTreePatterns

TreeJoin[*Axis*, *nt*](IN#in)	→	MapToItem{IN#out}(TupleTreePattern[IN#in/*Axis* :: *nt*{out}](IN))	(a)
ddo(MapToItem{TreeJoin[*Axis*, *nt*](IN#in)}(Op))	→	MapToItem{IN#out}(TupleTreePattern[IN#in/*Axis* :: *nt*{out}](Op))	(b)

Eliminating Item-Tuple Conversions

MapFromItem{[out1:IN]}
(MapToItem{IN#out2}
(TupleTreePattern[IN#in/*Axis* :: *nt*{out2}](Op)))
	→	TupleTreePattern[IN#in/*Axis* :: *nt*{out1}](Op)	(c)

Merging Individual Tree Patterns

TupleTreePattern[IN#out1/$step_2${out2}] (TupleTreePattern[IN#in/$pattern$/$step_1${out1}](Op))	→	TupleTreePattern[IN#in/$pattern$/$step_1$/$step_2${out2}](Op)	(d)

Select{
fn:boolean(fs:ddo(
(MapToItem{IN#out1}
(TupleTreePattern[IN#out/$pred_1${out1}] (IN))))
and
...
fn:boolean(fs:ddo(
(MapToItem{IN#outN}
(TupleTreePattern[IN#out/$pred_N${outN}] (IN))))
}
(TupleTreePattern[IN#in/$step${out}](Op))
	→	TupleTreePattern[IN#in/$step${out}[$pred_1$] . . . [$pred_n$]](Op)	(e)

fs:ddo(MapToItem{IN#out} (TupleTreePattern[IN#in/$pattern$](Op)))	→	TupleTreePattern[IN#in/$pattern$](Op)	(f)

Figure 3. XPath algebraic rewritings

```
3.    [IN#dot/descendant::person
4.     [child::emailaddress]/child::name{out}]
5.    (MapFromItem{[dot : IN]}($d)))
```

Note that the outer fs:ddo call is the subject of a very simple cleanup rule, since TupleTreePatterns incorporate tits semantics. Obviously, this example works all the way to the point where the original path expression is recovered. However, the benefit of that approach is that the query plans generated through those rewritings are always possible evaluation plans. For more complex path expressions, such as **Q2** or **Q3**, or **Q4**, **Q5** which were given in Section 2, the optimizer will detect only a certain fragment of the plan, leaving intermediate maps as necessary to preserve the proper semantics.

5 XPath Algorithms Revisited

In this section, we review the physical algorithms used to implement the logical TupleTreePattern operator described in the previous Section. In addition to nested-loop joins (NLJoin), we focus on two of the most well-known XPath join algorithms: TwigJoin [4] with the improvements suggested in [15, 23], and Staircase Join (SCJoin). We implemented those algorithms, taking some of the specificities of our physical setup into account (DOM in main memory, along with in-memory name indices). We start by describing our physical setup, then review the various physical algorithms. We also described improvements over the original algorithms that are made possibles by properties of the tree pattern generated from path expressions.

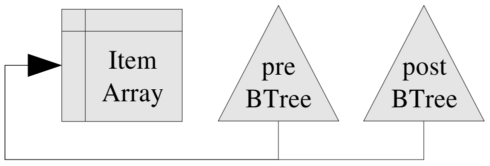

Figure 4. Name index for a given *QName*

5.1 Physical Data Model and Name Indexes

The main data structure we use to represent XML documents is an implementation of the XDM [28] based on a tree structure in main-memory. Each node contains local properties of that node (node-kind, node-name, typed-value, etc.), a pointer to its parent and a pointer to a list of its children.

In addition, we require the presence of name indices providing efficient access to element nodes based on pre-order and post-order in the tree. Access to both pre-order and post-order guarantees that the same index can be used for both the TwigJoin and the SCJoin algorithms.

Figure 4 shows one name index, which consists of (a) a main array of references to the DOM-elements with a matching QName in document order, (b) an index over the pre-order positions of the elements, pointing to the main array and (c) an index over the post-order positions of the elements, pointing to the main array. Note that any data structure providing efficient access to an element node given its pre-order or post-order number can be used. In our implementation, we use sorted arrays and binary search for efficient lookup. The array-based indexes were selected because they have the least overhead at build time, especially since we load them with already sorted key sequences, resulting in linear scalability with the document size.

5.2 Revisiting Nested Loops

The NLJoin algorithm is the default evaluation strategy when none of the other two algorithms can be applied. The biggest drawback of the nested loops physical algorithm is the need for sorting by document order and duplicate elimination. Duplicate elimination is especially important because it keeps the intermediate results from growing exponentially with duplicate nodes. On the other hand, duplicate elimination is a blocking operation and thus may significantly slow down query evaluation. Note however, that sorting and duplicate elimination operations can often be removed using static analysis techniques such as described in [13].

Note that even if a TupleTreePattern operator is introduced in the plan during logical optimization, TwigJoin and SCJoin algorithms may not be applicable. More precisely, the following constraints must hold:

- the tree pattern must be restricted to a case where the algorithms can be applied (e.g., no backward axis);
- the sequence of input nodes must be sorted by document order and duplicate free;
- appropriate name indexes must be available for the appropriate (parts of) the document.

Finally, note that NLJoin is more than a fall-back strategy. As we show in Section 6, NLJoins can be the best evaluation strategy for some specific queries, even if a TwigJoin or SCJoin is available.

5.3 Revisiting TwigJoins

Initial reports on TwigJoins [4] discuss two algorithms. PathStack is an algorithm that is specifically designed and optimized for straight-line tree patterns, i.e., tree patterns without branches. We do not consider this algorithm in this work and we restrict this discussion to the more generic counterpart of PathStack, namely TwigStack.

The TwigStack algorithm evaluates an entire tree pattern, including the branches, in one pass over the indexes. Our name indexes provide in-order access to all elements with the same QName and to their pre- and post-order positions, which is exactly what the TwigStack algorithm needs. The general idea behind TwigStack is to restrict query processing to complete matchings of the tree pattern only, skipping any partial and thus irrelevant results. For this, it maintains a stack for every tree pattern node, which keeps track of all root-to-leaf matchings of the tree pattern. Before a node n_q is pushed on a stack S_q, the algorithm ensures, by inspecting the corresponding QName indexes, that (i) there is a descendant of n_q in the index of all child tree pattern nodes and (ii) all these nodes recursively satisfy this property. The algorithm also ensures that if a node n comes before n' in document order, and if both n and n' are part of the result, that n is pushed on a stack before n'. Despite its optimality, TwigStack has some specific drawbacks:

- The algorithm executes a significant number of instructions for each node visited.
- TwigStack produces all separate root-to-leaf matches, which are later joined into full tree pattern matchings. This last step requires blocking, which is undesirable, especially for main memory query processors.

We use the above properties of the TwigStack algorithm, together with specifics of the XPath semantics to optimize TwigStack in several ways.

1. XPath queries only select nodes belonging to the stack of a single tree pattern node. We can use this property to remove the blocking factor from the TwigStack algorithm. Since we are not interested in the full tree pattern matching, there is no need for joining all partial results into full tree pattern answers. Note that for this optimization, we need to restrict the number of output fields inside the TupleTreePattern operators to one which is the cases where the semantics coincide with that of path expressions (see Section 3).
2. TwigStack ensures that every node that is pushed on any stack is part of the final result. Therefore, once an XPath output node is pushed on a stack, we do not need to match all descendant nodes on the descendant stacks, we can just skip them. Since the original algorithm ensures that nodes are pushed on the stacks in document order, we know that this modification causes the output to be sorted by document order and duplicate free.
3. Finally, we use the efficient lookup potential of the name indexes to optimize cursor navigation inside the TwigStack algorithm, as described in [15].

5.4 Revisiting Staircase Joins

The Staircase Join algorithm [18] is another well known evaluation strategy for XPath, which was originally designed to implement XPath using relational database technology. Staircase Joins are incorporating several optimizations that ensure the optimal evaluation of a single XPath step. The algorithm consists of the following steps:

- Pruning gets rid of context nodes whose results are subsets of those of other context nodes. This step avoids the occurrence of duplicate output nodes;
- Using pre-/post- order indexes, the algorithm identifies a continuous region in the relational table that encodes the nodes, where all the answers for one context node are located;
- A scanning of one completely isolated region per remaining context node results in the output list to be sorted and duplicate free;
- For some axes, optimality is attained by skipping over parts of the document regions that are known to be empty.

The key difference between our implementation and the original specification of the algorithm lies in the fact that Galax uses a DOM physical data layout, in contrast with the relational structure described in [18]. Since it is not desirable to duplicate the entire document in main memory in order to obtain the same indexing organization, we resort to using partial indexes, i.e., the name indexes described above. The disadvantage of this approach is that we cannot rely on the continuous addressing of the nodes based on the pre-order position of an item, a concept that is intensively used by the original Staircase Join algorithm to identify and scan partitions. As a consequence, the cost of identifying a region, including skipping, becomes $log(|Document|)$, as this requires an index lookup. This is in contrast with the constant cost of the original approach.

However, there is also an important advantage: the selectivity of the QName indexes will restrict scanning to nodes that match the name test of the step. Next to this, name indices are easier to maintain compared to a direct addressing system in case of updates. It is also important to point out that the main array of the indexes contains all items in document order, allowing sequential scanning within one region.

Finally, the original SCJoin algorithm does not handle predicate branches. In case the TupleTreePattern operator features such predicates, we evaluate the corresponding items on a per-node basis.

6 Experimental Evaluation

This section serves two purposes. First, we validate the robustness of the compilation approach described earlier in this paper. In order to do so, we run the compiler over semantically equivalent but syntactically different expressions and verify that appropriate tree patterns are detected.

In the second part of this section we derive heuristics for deciding among XPath join algorithms by comparing the relative performance of several popular tree pattern algorithms in the context of complete query plans (as opposed to in isolation). For convenience, we briefly review the main findings of that analysis here:

- For simple rooted path expressions, NLJoin is always outperformed by SCJoin or TwigJoin. SCJoins and TwigJoins are often providing very comparable performances, differing only by a constant factor;
- The performance of SCJoin can degrade for complex tree patterns while TwigJoin is always well-behaved;
- There is no single best algorithm for evaluating tree pattern operators in a query plan. A combination of parameters, including the form of the query and the shape and size of the documents must be taken into account to predict which XPath join algorithms performs best. Clearly, an accurate cost model is needed.

6.1 Validation of the logical rewritings

As we have seen, one of the benefits of the proposed compilation pipeline is the ability to detect tree patterns even in cases where navigation is not written directly in XPath, but within a combination of FLWOR and path expressions. Consider for instance the following path expression.

```
$input/site/people/person
   [emailaddress]/profile/interest
```

There are many equivalent variations of that path expression, such as the following FLWOR expression.

```
for $x1 in $input/site,
    $x2 in $x1/people,
    $x3 in $x2/person[emailaddress]
return $x3/profile/interest
```

In order to test the robustness of our rewritings[3], we generated 20 variants of the above path expression by replacing the / operator by equivalent `for` clauses and optionally

[3]Some of these syntactic variations require static analysis in order to decide that intermediate results are in document order and free of duplicates (see [25]).

replacing the predicate by a `where` clause. We ran these queries both on the standard engine (with no TupleTreePattern operator) and on the new engine. While on the old engine the generated plans were dependent on the syntactic form of the query, on the new engine, all the variants generated the exact same plan, containing a single TupleTreePattern operator.

The recovery of the path expression enables the application of specialized XPath algorithms, which in turn speeds up query evaluation and improves overall scalability, as is shown in Figure 5.

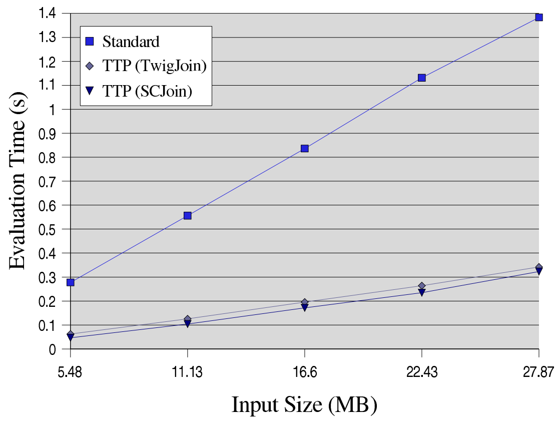

Figure 5. Evaluation of a path expression written as a FLWOR, with and without the rewrites

		2.1 MB	4.3 MB	6.5 MB	8.7 MB	11 MB
	NL	0.0661	0.14784	0.2137	0.3078	0.3856
QE1	TJ	0.0207	0.04369	0.0698	0.1264	**0.1468**
	SC	**0.0150**	**0.04310**	**0.0645**	**0.1063**	0.1570
	NL	0.0698	0.1247	0.2193	0.3035	0.3557
QE2	TJ	0.0380	0.0686	0.1102	0.1531	0.2131
	SC	**0.0157**	**0.0463**	**0.0630**	**0.1091**	**0.1358**
	NL	0.0686	0.1431	0.2247	0.2952	0.4086
QE3	TJ	**0.0179**	**0.0586**	**0.0749**	**0.1177**	**0.1656**
	SC	0.0212	0.0589	0.1126	0.1473	0.2226
	NL	0.0668	0.1393	0.2259	0.3201	0.3864
QE4	TJ	**0.0205**	**0.0446**	**0.0772**	0.1151	0.1549
	SC	0.0206	0.0544	0.0828	**0.0957**	**0.1339**
	NL	0.0714	0.1412	0.2296	0.3038	0.3652
QE5	TJ	0.0343	0.0825	0.1081	0.1619	0.2783
	SC	**0.0207**	**0.0489**	**0.0622**	**0.1058**	**0.1541**
	NL	0.0701	0.1513	0.2334	0.3294	0.4136
QE6	TJ	**0.0182**	**0.0481**	**0.0799**	**0.1105**	**0.1506**
	SC	0.0203	0.0587	0.0832	0.1372	0.1651

Table 1. Evaluation time (in seconds) for the queries in Figure 6

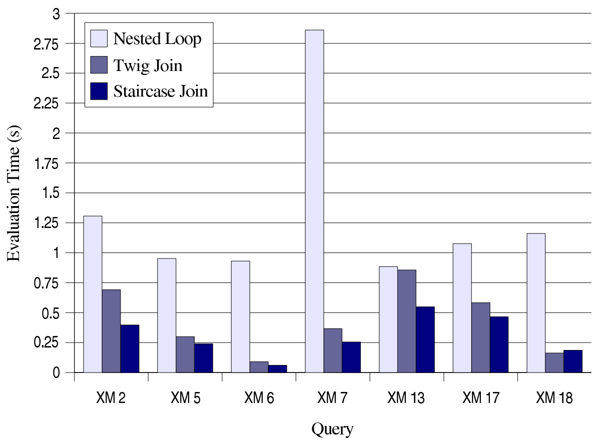

Figure 7. XMark queries where `child` has been replaced with `descendant`

6.2 Comparative Experiments

We ran experiments on both synthetic queries and the XMark benchmarks. We show that there is no single best XPath evaluation strategy and that the performance of the different algorithms depends on many variables, each favoring or disfavoring different aspects of individual algorithms.

For the first experiment we evaluate the six queries given in Figure 6, with all three evaluation strategies and different sizes of documents, namely 2.1, 4.3, 6.5, 8.7 and 11 MB MemBeR documents of depth 4, containing 100 different tags, uniformly distributed. The results of this test are shown in Table 1 (where we highlight the best times in boldface). Interestingly, it is sometimes beneficial to turn some child steps into descendant steps to benefit from the improved handling of descendant in SCJoin and TwigJoin. Figure 7 show the evaluation time for several XMark queries for which replacing a `child` by a `descendant` step without changing the semantics.

The results of both tests show that NLJoin is never the fastest strategy. The reason for this lies in the construction of both queries and documents. However, for more complex path expressions, NLJoin can sometimes be the fastest algorithm.

The XMark results indicate that SCJoins are the better evaluation strategy in most cases. These and other experiments have shown that SCJoins indeed provide good performance and scalability for simple path expressions. The synthetic queries QE1 to QE6, however, point out that, once the query gets sufficiently complex, SCJoins are no longer the fastest. The results for QE2 and QE5 in Table 1 show that TwigJoins are clearly faster. But query complexity is obviously not the only factor affecting performance, as is shown in the results for QE1 and QE4, where the document size seems to influence which algorithm performs best.

It is also worth pointing out that evaluating child axes does not penalize query performance in both TwigJoin and

(QE1) `$input/desc::t01[child::t02[child::t03[child::t04]]]`
(QE2) `$input/desc::t01/child::t02[1]/child::t03[child::t04]`
(QE3) `$input/desc::t01[child::t02[child::t03]/child::t04[child::t03]]`
(QE4) `$input/desc::t01[desc::t02[desc::t03[desc::t04]]]`
(QE5) `$input/desc::t01/desc::t02[1]/desc::t03[desc::t04]`
(QE6) `$input/desc::t01[desc::t02[desc::t03]/desc::t04[desc::t03]]`

Figure 6. The first three queries correspond with the last three, but all `child` axes (except the first) have been replaced with `descendant` ones.

SCJoin algorithms. This is due to the constant access cost to children and parent in the Galax data model.

6.3 XPath Evaluation in an XQuery Context

In previous experiments, involving isolated tree pattern expressions, NLJoins are slower than either TwigJoins and SCJoins. This is not always the case for more complex path expressions or path expressions within large more complex queries. This experiment shows that once path expressions fall outside the tree pattern fragment, or are embedded inside XQuery expressions, the relative performance among the join algorithms changes.

Experiment Setup – We used a MemBeR document of 50,000 nodes and depth 15. All nodes have the same qname t1. Next, we evaluated the queries $(\texttt{/t1[1]})^k$ for $k = 5, 10, 15$. The results are as follows:

	$k = 5$	$k = 10$	$k = 15$
NLJoin	**0.00683**	**0.00064**	**0.00059**
TwigJoin	0.85847	0.88072	0.84561
SCJoin	0.23770	0.23803	0.21785

There are a few reasons for the big difference between NLJoin on the one hand and TwigJoins and SCJoins on the other.

- Since the query falls outside the tree pattern fragment, the plan will contain TupleTreePattern operators embedded in maps. So, both TwigJoins and SCJoins will scan the index once for each step,
- The query is very selective, causing Nested Loop to visit a very limited portion of the tree.
- The NLJoin only accesses the first child of every step, because of the cursor based implementation. The other algorithms have a cost of at least $Log(|Input|)$ per step for the index lookup.

Although this is a quite extreme example, it suffices to achieve sufficient selectivity in a query in order for NLJoins to benefit from this.

7 Related work

In the past several years, efficient XML processing has received considerable interest. Numerous efforts have focused on the development of algorithms [4, 15, 23, 19, 9, 18, 16], along with appropriate index structures [17, 8, 21, 7, 5, 20, 22], based on the notion of tree pattern has been a key focus of research on XML query processing. Our work is essentially complementary, as it aims at bridging the gap between algorithmic work on tree patterns and XML algebraic compilers.

The first complete algebraic treatment of XPath 1.0 was proposed in [3]. The approach uses a nested-relational algebra which enables well-understood relational optimizations, including traditional join optimization. However, it does not integrate support for tree-pattern operators as presented here. The TAX algebra [6] developed for the Timber system is a tree-based algebra that support tree-pattern evaluation. Compilation for a fragment of XQuery into TAX is described in [6]. However it is unclear how the approach can scale to arbitrary XQuery expressions, and how to integrate traditional relational optimization in the context of a purely tree-based approach. Our work is the first to extend a tuple algebra for XQuery [27] with support for tree-pattern evaluation. Work on System RX [1] includes a tree pattern operator capable of returning multiple bindings. However, the compiler relies on the syntactic form of the path expressions to detect when a tree pattern operator can be used. Instead, we provide a more semantic treatment which facilitates the recognition of tree pattern operators in more complex queries.

8 Conclusion

In this paper, we have proposed an approach that support the systematic compilation of path expression in arbitrary XQuery. We developed algebraic techniques to detect the possible use of tree-pattern operators, and presented the first comparative evaluation between several well known XPath algorithms. Our experiments indicate that there is no single best algorithm for evaluating path expressions in the context of XQuery, and point to the necessity of developing a suitable cost-model.

Interesting future work include the extension of the tree-pattern fragment that is being suported to deal with positional predicates. We are also interested in evaluating the benefits of other variants of Twigjoin algorithms, as well as the possible use of streaming XPath algorithms. Also, our study so far has been limited to a single data model. We are considering the use of more advanced disc-based indices and extending our approach to specific shredding models such as the one proposed in [17].

Acknowledgments. We would like to thank Mary Fernández and Christopher Ré for numerous discussions on XPath evaluation, and their help with the implementation.

Philippe Michiels is supported by the IWT (Institute for the Encouragement of Innovation by Science and Technology) Flanders, grant number 31016. Some of this work was done during an internship at the IBM Watson Research Center (NY), partially funded by IBM.

References

[1] K. Beyer, R. J. Cochrane, V. Josifovski, J. Kleewein, G. Lapis, G. Lohman, B. Lyle, F. Özcan, H. Pirahesh, N. Seemann, T. Truong, B. V. der Linden, B. Vickery, and C. Zhang. System RX: one part relational, one part XML. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 347–358, New York, NY, USA, 2005.

[2] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML query language. Candidate Recommendation, Nov. 2005.

[3] M. Branter, S. Elmer, C.-C. Kanne, and G. Moerkotte. Full-fledged algebraic XPath processing in Natix. In *ICDE*, Boston, MA, 2005.

[4] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 310–321, New York, NY, USA, 2002. ACM Press.

[5] T. Chen, T. W. Ling, and C. Y. Chan. Prefix path streaming: A new clustering method for optimal holistic XML twig pattern matching. In *DEXA*, pages 801–810, 2004.

[6] Z. Chen, H. V. Jagadish, L. V. S. Lakshmanan, and S. Paparizos. From tree patterns to generalized tree patterns: On efficient evaluation of XQuery. In *VLDB*, pages 237–248, Berlin, Germany, Sept. 2003.

[7] S. Chien, Z. Vagena, D. Zhang, V. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In *VLDB*, Hong Kong, China, Aug. 2002.

[8] J.-K. M. Chin-Wan Chung and K. Shim. APEX : An adaptive path index for XML data. *SIGMOD*, 15(5):121–132, June 2002.

[9] B. Choi, M. Mahoui, and D. Wood. On the optimality of holistic algorithms for twig queries. In *DEXA*, pages 28–37, 2003.

[10] D. Colazzo, G. Ghelli, P. Manghi, and C. Sartiani. Types for path correctness of xml queries. In *ICFP*, pages 126–137, 2004.

[11] A. Deutsch, Y. Papakonstantinou, and Y. Xu. The NEXT logical framework for XQuery. In *VLDB*, pages 168–179, Toronto, Canada, Aug. 2004.

[12] D. Draper, P. Fankhauser, M. Fernandez, A. Malhotra, K. Rose, M. Rys, J. Simeon, and P. Wadler. XQuery 1.0 and XPath 2.0 formal semantics, W3C working draft. Candidate Recommendation, Nov. 2005.

[13] M. Fernández, J. Hidders, P. Michiels, J. Siméon, and R. Vercammen. Optimizing sorting and duplicate elimination in XQuery path expressions. In *Proc. of the 16th International Conference on Database and Expert Systems Applications (DEXA 2005)*, volume 3588 of *Lecture Notes in Computer Science*, pages 554–563, Copenhagen, Denmark, 2005.

[14] M. Fernández, J. Siméon, B. Choi, A. Marian, and G. Sur. Implementing XQuery 1.0: The Galax Experience. In *VLDB*, Sept. 2003.

[15] M. Fontoura, V. Josifovski, E. Shekita, and B. Yang. Optimizing cursor movement in holistic twig joins. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 784–791, New York, NY, USA, 2005. ACM Press.

[16] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *VLDB*, pages 95–106, 2002.

[17] T. Grust, M. V. Keulen, and J. Teubner. Accelerating XPath evaluation in any RDBMS. *ACM Trans. Database Syst.*, 29(1):91–131, 2004.

[18] T. Grust and M. van Keulen. Tree awareness for relational DBMS kernels: Staircase join. In *Intelligent Search on XML Data*, pages 231–245, 2003.

[19] H. Jiang, H. Lu, and W. Wang. Efficient processing of twig queries with or-predicates. In *SIGMOD Conference*, pages 59–70, 2004.

[20] H. Jiang, W. Wang, H. Lu, and J. X. Yu. Holistic twig joins on indexed XML documents. In *VLDB*, pages 273–284, 2003.

[21] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *VLDB*, September 2001.

[22] J. Lu, T. W. Ling, C. Y. Chan, and T. Chen. From region encoding to extended Dewey: On efficient processing of XML twig pattern matching. In *VLDB*, pages 193–204, 2005.

[23] J. Lu, T. W. Ling, T. Yu, C. Li, and W. Ni. Efficient processing of ordered XML twig pattern. In *DEXA*, pages 300–309, 2005.

[24] N. May, S. Helmer, and G. Moerkotte. Nested queries and quantifiers in an ordered context. In *ICDE*, pages 239–250, Boston, MA, Mar. 2004.

[25] P. Michiels, G. A. Mihăilă, and J. Siméon. Put a tree pattern in your tuple algebra. Technical report, Tech. Rep., Univ. of Antwerp, TR-06-09, Belgium, 2006.

[26] S. Paparizos, Y. Wu, L. V. S. Lakshmanan, and H. V. Jagadish. Tree logical classes for efficient evaluation of XQuery. In *SIGMOD*, pages 71–82, Paris, France, June 2004.

[27] C. Re, J. Simeon, and M. Fernandez. A complete and efficient algebraic compiler for XQuery. In *ICDE*, Atlanta, Georgia, Apr. 2006.

[28] XQuery 1.0 and XPath 2.0 data model (XDM). Candidate Recommendation, Nov. 2005.

A APPENDIX: A Normal Form for Tree Patterns in XQuery

We conjecture that our complete set of rewrite rules (given below) is capable of picking up all expressions inside the supported fragment that have XPath semantics and rewrite them into a Tree Pattern Normal Form such that they can be picked up by the algebraic rewrites as TupleTreePattern operators. Due to time restrictions, the formal proof for this conjecture could not be verified and written down in this paper and it is left as future work.

A.1 Supported Frgament

$expr$ ::= \$$x$ | . | * | l | $expr$/$expr$ | $expr$//$expr$ | $expr$[$expr$] |
(let \$$x$:= $expr$ | for \$$x$ in $expr$)* (where $cond$)? return $expr$
$cond$ ::= $expr$ | $cond$ and $cond$

A.2 XQuery Core Mapping

A.2.1 Core Fragment (CXQ)

$expr$ ::= \$$x$ | \$$x$/child::* | \$$x$/child::$l$ | \$$x$/descendant-or-self::* |
fs:distint-docorder($expr$) | for \$$x$ in $expr$where $cond$ return $expr$
$cond$ ::= $expr$ | $cond$ and $cond$

A.2.2 Mapping Rules

Conjecture 1 *Every XQ expression e is mapped to an equivalent expression e' in CXQ, using the translation rules below.*

Slash Normalization

```
e1/e2   ↦   fs:distinct-docorder(
               for $fs:dot in e1
               return e2
            )
```

```
e1//e2   ↦   fs:distinct-docorder(
                for $fs:dot in e1/descendant-or-self::*
                return e2
             )
```

Predicate Normalization

```
e1[e2]   ↦   for $fs:dot in e1
             where e2
             return $fs:dot
```

Inlining

```
let $x := e1
return e2     ↦     e2[$x/e1]
```

Steps

$$. \mapsto \text{\$fs:dot}$$

$$l \mapsto \text{child::}l$$

$$* \mapsto \text{child::*}$$

Introducing Where

$$\begin{array}{l}\text{for \$x in } e_1\\ \text{return } e_2\end{array} \quad \mapsto \quad \begin{array}{l}\text{for \$x in } e_1\\ \text{where \$x}\\ \text{returnn } e_2\end{array}$$

A.3 Tree Pattern Normalization rules

A.3.1 Tree Pattern Normal Form (TPNF)

The language TPNF is a sublanguage of CXQ and is defined by following grammar:

$$\begin{array}{ll} expr ::= & for \mid step \mid self \\ for ::= & \text{fs:distint-docorder(for \$fs:dot in } expr \text{ where } expr \text{ return } step\text{)} \\ step ::= & \text{\$fs:dot/child::*} \mid \text{\$fs:dot/child::}l \mid \text{\$fs:dot/descendant-or-self::*} \\ self ::= & \text{\$fs:dot} \end{array}$$

A.4 TPNF Normailization

Below we give the complete set of rewrite rules that are needed for rewriting any expression in CXQ that has XPath semantics into TPNF.

A.4.1 Pre-processing

Insert fs:distinct-docorder Call

$$\begin{array}{l}\text{fs:distinct-docorder(}\\ \quad \text{for \$x in } e_1\\ \quad \text{where } e_2\\ \quad \text{return } e_3\\ \text{)}\end{array} \quad \mapsto \quad \begin{array}{l}\text{fs:distinct-docorder(}\\ \quad \text{for \$x in fs:distinct-docorder(}e_1\text{)}\\ \quad \text{where } e_2\\ \quad \text{return fs:distinct-docorder(}e_3\text{)}\\ \text{)}\end{array}$$

With e_1, e_3 not a distinct-docorder call.

Remove Duplicate fs:distinct-docorder Calls

$$\text{fs:distinct-docorder(fs:distinct-docorder(}e\text{))} \quad \mapsto \quad \text{fs:distinct-docorder(}e\text{)}$$

Resulting grammar:

$$\begin{array}{ll} expr ::= & \text{fs:distinct-docorder(\$}x \mid \text{\$}x\text{/child::*} \mid \text{\$}x\text{/child::}l \mid \text{\$}x\text{/descendant-or-self::*} \mid \\ & \text{for \$}x \text{ in } expr\text{where } cond \text{ return } expr \text{)} \\ cond ::= & expr \mid cond \text{ and } cond \end{array}$$

A.4.2 Actual Normalization Rules

Loop Fusion

```
fs:distinct-docorder(                      fs:distinct-docorder(
   for $x in $e_1$                            for $y in
   where $c_2$                                   fs:distinct-docorder(
   return                                           for $x in $e_1$
      fs:distinct-docorder(     ↦                   where $c_2$
         for $y in $e_3$                            return $e_3$ )
         where $c_4$                          where $c_4$
         return $e_5$                         return $e_5$)
      ))
```

If $x $\notin FV(c_4) \cup FV(e_5)$.

Lift Condition

```
fs:distinct-docorder(                      fs:distinct-docorder(
   for $x in $e_1$                            for $x in $e_1$
   where $c_2$                                where $c_2$ and $c_4$
   return                                     return
      fs:distinct-docorder(     ↦                fs:distinct-docorder(
         for $y in $e_3$                            for $y in $e_3$
         where $c_4$                                return $e_5$
         return $e_5$                            ))
      ))
```

Where $y $\notin FV(c_4)$.

```
fs:distinct-docorder(                      fs:distinct-docorder(
   for $y in                                  for $y in
      fs:distinct-docorder(                      fs:distinct-docorder(
         for $x in $e_1$                            for $x in $e_1$
         where $c_2$            ↦                   where $x
         return $e_3$                               return $e_3$
      )) where $c_4$                             )) where $c_4$ and $c_2$
   return $e_5$                               return $e_5$
```

Where $x $\notin FV(c_2)$.

Extract Condition

```
                                           fs:distinct-docorder(
                                              for $z in $y
      fs:distinct-docorder(                   where
         for $x in $e_1$                         for $x in $e_1$
         where $c_2$            ↦                where $c_2$
         return $y/$step$                        return $x
      )                                       return $z/$step$
                                           )
```

When $x $\neq$ $y. The *step* is optional.

A.4.3 Post-processing

Predicate Decomposition

```
for $x in $e$                     for $x in
where $e_1$ and $e_2$    ↦            for $x in $e$
return $e_r$                          where $e_1$
                                      return $x
                                  where $e_2$
                                  return $e_r$
```

Redundant Loop Removal

for \$x in e where \$x return \$x $\mapsto$ e

Remove fs:distinct-docorder Calls

fs:distinct-docorder(\$$x$) $\mapsto$ \$$x$

fs:distinct-docorder(\$$x$/child::*) $\mapsto$ \$$x$/child::*

fs:distinct-docorder(\$$x$/child::$l$) $\mapsto$ \$$x$/child::$l$

fs:distinct-docorder(\$$x$/descendant-or-self::*) $\mapsto$ \$$x$/descendant-or-self::*

A.4.4 Identifying Implicit Tree Patterns

Insert fs:distinct-docorder Call (bis)

e $\mapsto$ fs:distinct-docorder(e)

When the result of e always is in document order of free from duplicates, independantly of the input.

Note that for the last rule we need to verify wether an expression always yields a result that is sorted by document order and duplicate-free. We conjecture here, that this can indeed be decided for the CXQ fragment.

The latter rule detects all non XPath expressions that have XPath semantics and puts an fs:distinct-docorder call around them. Proving completeness is done by showing that the above rules rewrite all expressions that are surrounded by an fs:distinct-docorder call to expressions in TPNF.