Put a Tree Pattern in Your Algebra

Philippe Michiels
University of Antwerp
philippe.michiels@ua.ac.be

George A. Mihăilă
IBM Research
mihaila@us.ibm.com

Jérôme Siméon
IBM Research
simeon@us.ibm.com

Abstract

To address the needs of data intensive XML applications, a 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 XML algebras, but also because deciding 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 that operator in query plans. We demonstrate the robustness of the proposed rewritings under syntactic variations commonly found in queries. The proposed tree pattern operator can be implemented using popular algorithms such as Twig joins and Staircase joins. Our experiments yield useful information to decide which algorithm should be used in 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, 24, 20, 8, 17, 16]. Most XQuery algebras [11, 25, 28, 13, 1] do not support tree patterns, focusing instead on recovering important relational optimizations [11, 25, 28, 13, 1] or on how to support larger fragments of the language [3, 28]. 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, 27] supports tree patterns, but the absence of support for tuple operators makes it difficult to recover important relational optimizations. System RX [1] is the only algebra to support both relational rewritings and a tree pattern operator. However, it recognizes tree patterns based on syntax, while we provide a more semantic treatment which enables the recognition of tree pattern operators in complex queries. More precisely, we answer three important questions: how can tree patterns be integrated in a tuple algebra for XQuery, how can the compiler decide when a part of a query plan can be evaluated using a tree pattern, and how to decide which tree pattern algorithm to use.

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 an XQuery compiler to detect when efficient tree pattern algorithms can be used. It is important to stress that we limit our discussion to tree patterns with a single output node, i.e., corresponding to a single XPath expression. Extending our approach for multiple-variable tree patterns is kept for fututre work. 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 rewriting 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 [13][1] and experiments show the robustness of the approach for complex

---

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

1-4244-0803-2/07/$20.00 ©2007 IEEE.

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

tree pattern queries.

The main contributions of the paper are as follows:

- We extend the algebra of [28] with a tree pattern operator. That operator is designed to integrate with tuple operators and can be implemented as Twig joins [4, 15] or Staircase joins [17].
- 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 equivalences 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, yielding useful information to decide which algorithm should be used in a given plan.

The rest of the paper is organized as follows. Section 2 gives an overview of our compilation approach. Section 3 describes the rewritings used to normalize tree patterns in arbitrary queries. Section 4 presents the algebraic tree pattern operator and the algebraic equivalences. Section 5 contains the experimental evaluation. Section 6 reviews the related work, and Section 7 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, 28]. The normalization rules for path expressions (corresponding to the following grammar productions (68), (69), (70), (71), and (81) in [2]), are described in detail in [12]. For instance, normalization applied to **(Q1a)** on Figure 1 results in the following XQuery Core expression[2].

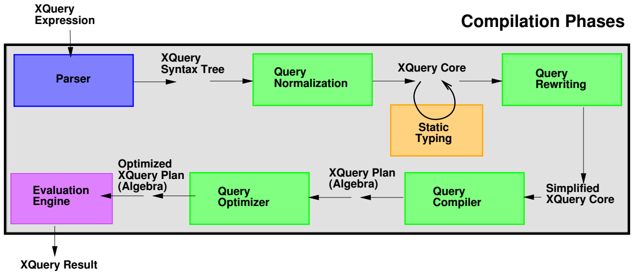

Figure 2. Compilation Architecture

Q1a-n
```
1. ddo(
2.  let $seq :=
3.   ddo(
4.    let $seq := ddo($d),
5.    let $last := fn:count($seq)
6.    for $dot at $position in $seq
7.    return
8.     let $seq := ddo(descendant::person),
9.     let $last := fn:count($seq)
10.    for $dot at $position in $seq
11.    where
12.     typeswitch (child::emailaddress)
13.      case $v2 as numeric() return
14.       $position = $v2
15.      default $v3 return
16.       fn:boolean($v3)
17.    return $dot),
18. let $last := fn:count($seq)
19. for $dot at $position in $seq
20. return child::name)
```

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

---

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

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`[3] 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 $E_1[E_2]$ expressions. 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 rewriting, 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**.

Q1-tp

```
ddo(
 for $dot in
   for $dot in
    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 [28].

P1

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

For conciseness, the query plans are written in the functional notation used in [28], 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

---

[3] `ddo` is a shorter name for `fs:distinct-doc-order` [12].

see in Section 5 that deciding which algorithm to use is non trivial, and that there is a need for a cost based approach for evaluating XPath expressions, which adds to the importance of identifying tree patterns.

3 Tree Pattern Rewritings

In this secion, we briefly discuss the rewriting techniques needed to prepare XQuery expressions for tree pattern detection. To illustrate the effect of the 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, consisting of fairly straightforward logical optimizations and a set of normalizing rewrites. The full set of rewritings normalizes XQuery expressions into a form called TPNF [19]. The set of rules is shown to be complete for a substantial fragment of the XQuery language and applying the rules exhaustively always terminates. Implementation details of the Galax XQuery engine lead us to use a slightly different set of rules than those presented in [19]. We conjecture that this rule set, which leads to a different normal form called TPNF' in turn is complete for the presented fragment. However, a formal study of the properties of those rewritings is beyond the scope of this paper and the intested reader is referred to [19]. The experimental evaluation in Section 5 confirms the robustness of the 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)
```

Document order rewritings. As described in detail in [19], we can identify tree pattern equivalent expressions by deciding which subqueries always produce results in document order and free from duplicates. By introducing and propagating annotations, we can remove all but the surrounding sorting operations. The full details of the removal of these redundant `ddo` operations are given in [19]. The result of removing the unnecessary sorting operations is as follows.

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

After this rewriting, the expressions surrounded by a sorting operation are known to be equivalent to tree patterns.

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

1-4244-0803-2/07/$20.00 ©2007 IEEE.

```
for $x in $Expr_1$ (where $Cond_1$)? return
  for $y in $Expr_2$ (where $Cond_2$)? return $Expr_3$
-------------------------------------------------------
for $y in
  for $x in $Expr_1$ (where $Cond_1$)? return $Expr_2$
(where $Cond_2$)? return $Expr_3$
```

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

```
ddo(
 for $dot in
  for $dot in
   for $dot in $d
   return 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]`.

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

Applying loop splitting 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. Applying the full set of 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 [28] 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.

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

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) [14], $TreePattern\{q_1,...,q_n\}$ denotes a tree pattern containing the output fields $q_1,...,q_n$.

$$\text{TupleTreePattern}[TreePattern\{q_1,...,q_n\}](\texttt{S}(\tau)) \\ \rightarrow \\ \texttt{S}([q_1{:}\texttt{S}(i);...;q_n{:}\texttt{S}(i)])$$

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"/></c></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 again the path expression **Q1a**. As we have seen in the previous Sections, applying the normalization and rewriting phases results in the expression **Q1-tp** which is then compiled in the algebra by applying the compilation rules in [28], resulting in the plan **P1**. This plan features map operators, TreeJoin, and `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 TupleTreePatterns 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.

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). 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.       (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.            (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. Applying rule (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.       (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 that occur next to each other into multi-step tree patterns. 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 in the plan **P4** below, in which the Select operation to be removed and a predicate branch added to the tree pattern.

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

Note that the outer fs:ddo call is the subject of a very simple cleanup rule (f), since TupleTreePatterns incorporate its 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

Replacing TreeJoins with TupleTreePatterns

TreeJoin[*Axis*, *NT*](IN#in)	→	MapToItem{IN#out}(TupleTreePattern[IN#in/*Axis* :: *NT*{out}](IN))	(a)
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(
(MapToItem{IN#out1}
(TupleTreePattern[IN#out/$pred_1${out1}] (IN)))
and
. . .
fn:boolean(
(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)))	→	MapToItem{IN#out} (TupleTreePattern[IN#in/$pattern$](Op))	(f)

Figure 3. XPath algebraic rewritings

Section 2, the optimizer will detect only a certain fragment of the plan, leaving intermediate maps as necessary to preserve the proper semantics.

The rewrites presented above can be seen as a mapping from TPNF' expressions to tree patterns. We prove that this mapping indeed rewrites all tree patterns from TPNF into an expression of the form:

```
MapToItem{IN#out}                              P5
  (TupleTreePattern
    [ PATTERN{out}]
    (MapFromItem{[dot : IN]}($v)))
```

The proof is given in the technical report [26]. Thus, the set of rewrites presented here always finds the largest tree pattern within the supported XQuery fragment.

5 Experimental Evaluation

This section serves two purposes. First, we validate the robustness of the compilation approach described in the 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:

- In the case of simple rooted path expressions, Nested Loop Structural Join (NLJoin) is always outperformed by Staircase Join (SCJoin) [17] or Holistic Twig Join (TwigJoin) [4]. 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.

5.1 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
```

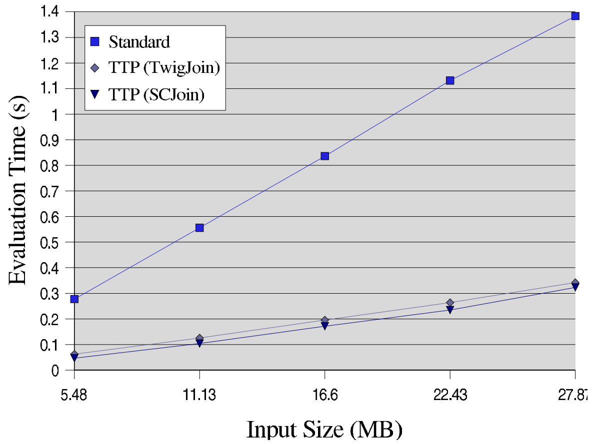

Figure 4. Evaluation of a path expression written as a FLWOR, with and without the rewrites

	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 5

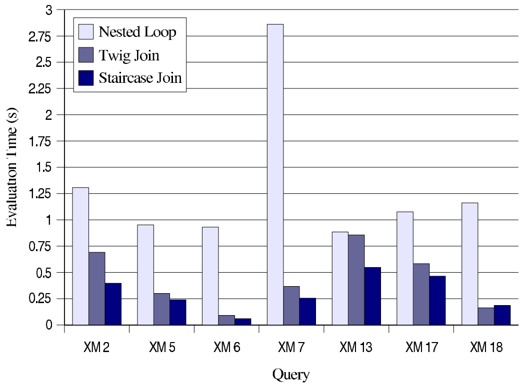

Figure 6. XMark queries where `child` has been replaced with `descendant`

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
```

These variations require further analysis in order to decide whether intermediate results are in document order and duplicate free [19, 26]. We generated 20 variants of the above path expression by replacing the / operator by equivalent `for` clauses and optionally 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 4.

5.2 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 5, 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 6 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 queries, 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

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

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

5.3 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.

6 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, 24, 20, 8, 17, 16], along with appropriate index structures [18, 9, 22, 7, 5, 21, 23], based on the notion of tree pattern. Our work is essentially complementary, as it aims at bridging the gap between algorithmic work on tree patterns and algebraic XQuery 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 [28] 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.

7 Conclusion

In this paper, we have proposed an approach that supports the systematic detection and compilation of tree patterns in arbitrary XQuery. We developed algebraic techniques to introduce tree-pattern operators in query plans based on a tuple-based algebra. The approach is shown to be robust for complex syntactic variations found in queries. Our experiments suggest heuristics to decide which algorithm to use for tree patterns in simple plans, but also the necessity of developing a suitable cost-model for more complex queries.

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 disk-based indices and extending our approach to specific shredding models such as the one proposed in [18].

Acknowledgments. We want to thank Jan Hidders for his help on the formal treatment of the rewritings, and Mary Fernández and Christopher Ré for feedback on earlier versions of the paper.

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. 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*, pages 347–358, 2005.

[2] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML query language. W3C Candidate Recommendation, June 2006.

[3] M. Brantner, S. Helmer, C.-C. Kanne, and G. Moerkotte. Full-fledged algebraic xpath processing in natix. In *ICDE*, pages 705–716, 2005.

[4] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal xml pattern matching. In *SIGMOD*, pages 310–321, 2002.

[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, 2003.

[7] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed xml documents. In *VLDB*, pages 263–274, 2002.

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

[9] C.-W. Chung, J.-K. Min, and K. Shim. Apex: an adaptive path index for xml data. In *SIGMOD*, pages 121–132, 2002.

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

[14] M. F. Fernández, A. Malhorta, J. Marsh, M. Nagy, and N. Walsh. XQuery 1.0 and XPath 2.0 data model (XDM), July 2006. .

[15] M. Fontoura, V. Josifovski, E. Shekita, and B. Yang. Optimizing cursor movement in holistic twig joins. In *CIKM*, pages 784–791, 2005.

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

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

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

[19] J. Hidders, P. Michiels, J. Siméon, and R. Vercammen. How to recognize different kinds of tree patterns from quite a long way away. Technical Report TR UA 13-2006, Univ. of Antwerp and IBM Research, 2006. http://www.adrem.ua.ac.be.

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

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

[22] Q. Li and B. Moon. Indexing and querying xml data for regular path expressions. In *VLDB*, pages 361–370, 2001.

[23] 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.

[24] 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.

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

[26] P. Michiels, G. A. Mihăilă, and J. Siméon. Put a tree pattern in your tuple algebra. Technical Report TR UA 09-2006, Univ. of Antwerp and IBM Research, Belgium, 2006. http://www.adrem.ua.ac.be.

[27] 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, 2004.

[28] C. Re, J. Siméon, and M. F. Fernández. A complete and efficient algebraic compiler for xquery. In *ICDE*, page 14, 2006.