Optimizing Correlated Path Queries in XML Languages

Ning Zhang and M. Tamer Özsu*

Technical Report CS-2002-36 November 2002

University of Waterloo

*School Of Computer Science, University of Waterloo, {nzhang,tozsu}@uwaterloo.ca

Abstract

Path expressions are ubiquitous in XML processing languages such as XPath, XQuery, and XSLT. Expressions in these languages typically include multiple path expressions, some of them correlated. Existing approaches evaluate these path expressions one-at-a-time and miss the optimization opportunities that may be gained by exploiting the correlations among them. In this paper, we address the evaluation and optimization of correlated path expressions. In particular, we propose two types of optimization techniques: integrating correlated path expressions into a single pattern graph, and rewriting the pattern graph according to a set of rewriting rules. The first optimization technique allows the query optimizer to choose an execution plan that is impossible by using the existing approaches. The second optimization technique rewrites pattern graphs at a logical level and produce a set of equivalent pattern graphs from which a physical optimizer can choose given an appropriate cost function. Under certain conditions that we identify, the graph pattern matching-based execution approach that we propose may be more efficient than the join-based approaches.

1 Introduction

Path expressions are ubiquitous in XML processing languages (XPath [1], XQuery [2], XSLT [3], XPointer [4], to name a few). With a syntax that involves slash "/" separated, UNIX directory-like notation, path expressions are used to locate elements in a collection of XML documents. For example, path expression "`/bib/book/authors`" locates in the bibliography all authors who wrote a book; "`/bib/*[authors='John Smith']`" locates all of John Smith's publications (including those co-authored with others) in the bibliography.

Since path expressions are usually used in conjunction with other query languages such as XPath and XQuery, there could be multiple path expressions correlated with each other in a single XPath or XQuery expression. Consider the example query in Figure 1 taken from XQuery Use Cases [5]. There are four path expressions bound to four variables in the `for` clause, and ten more path expressions introduced by referencing the four variables in the `where` clause. Note that these path expressions are not independent, but are correlated with each other by variable referencing, simple equality test (=), substring containment test (`contains` function), and equality test on aggregation (`max`) of subexpressions.

Previous research on the evaluation and optimization of path expressions has focused on a single path expression [6, 7, 8, 9, 10], and follow two approaches. A straightforward approach is to follow the formal semantics of XPath and XQuery languages [11], and treat a path expression as a sequence of steps, each of which takes input from the previous step and produces an output to the next step. Although this approach seems simple, experimental results show that implementations following this approach suffer from exponential runtime in the size the of path expressions in the worst case [10].

Another approach is to reduce the problem of evaluating path expressions against XML documents to the problem of matching *pattern trees* against *subject trees*, where the subject tree is a simplified data model for an XML document, and the pattern tree is an abstraction for a fragment of path expression. This problem is called the *tree pattern matching* (TPM) problem. Intuitively, the TPM problem is to find all mappings δ_i from the nodes of the pattern tree to the nodes of the subject tree, such that δ_i is label-preserving and relationship-preserving (child-parent, descendant-ancestor, and sibling-ordering relationships). Using proper labeling techniques [6, 7, 8, 12, 13], TPM can be evaluated reasonably efficiently. However, it has not been shown that a complete path expression (including all thirteen axes) can be translated into a pattern tree. In fact, we shall show that a complete path expression cannot be translated into a pattern *tree* but can be translated into a directed graph, which we call the *pattern graph*. More

```
1.  <result> {
2.    for $seller in document("users.xml")//user_tuple,
3.        $buyer in document("users.xml")//user_tuple,
4.        $item in document("items.xml")//item_tuple,
5.        $highbid in document("bids.xml")//bid_tuple
6.    where $seller/name = "Tom Jones"
7.      and $seller/userid = $item/offered_by
8.      and contains($item/description,"Bicycle")
9.      and $item/itemno = $highbid/itemno
10.     and $highbid/userid = $buyer/userid
11.     and $highbid/bid =
12.         max( for $x in document("bids.xml")
13.              //bid_tuple[itemno=$item/itemno]/bid
14.               return decimal($x) )
15.   return
16.     <jones_bike>
17.       { $item/itemno }
18.       { $item/description }
19.       <high_bid>{ $highbid/bid }</high_bid>
20.       <high_bidder>{ $buyer/name }</high_bidder>
21.     </jones_bike>
22.   sort by {itemno}
23. } </result>
```

Figure 1: A Sample XQuery Expression in XQuery Use Cases [5] (Q5 in the use case "R")

generally, we shall show that multiple path expressions can be translated into a pattern graph, which is the basis for our optimization of correlated path expressions.

Both of the previous approaches can be easily extended to handle correlated path expressions: evaluate each path expression one-by-one, and decide whether to materialize/cache the results of path expressions to variable bindings. However, they omit the connections between path expressions, thus missing the optimization opportunities based on these connections. For example, in Figure 1, the path expression `document("users.xml")//user_tuple` in line 2 is correlated to the path expression `$seller/name` in line 6 since the latter references the variable `$seller`, which is bound to the former path expression. Due to this correlation, the two path expressions, together with the comparison expression between the second path expression and "Tom Jones", can be rewritten to a single path expression `document("users.xml")//user_tuple[name="Tom Jones"]` Instead of evaluating the two path expressions one-at-a-time and then evaluating the comparison expression, the query optimizer can choose an access plan that evaluates `//user_tuple[name="Tom Jones"]` directly from XML document `users.xml`.

In this paper, we propose a new way of evaluating multiple path expressions by exploiting the correlations among them. Our optimization methodology is illustrated in Figure 2. We start by converting multiple correlated path expressions to a pattern graph. The identification of the correlated path expressions is outside the scope of this paper, and will be the subject of our future work. The pattern graph is then rewritten (using rules introduced in the paper) to generate a set of equivalent pattern graphs from which the physical optimizer can choose. In particular, one rewriting rule partitions the pattern graph into quasi-ordered pattern trees, which are passed to

a tree pattern matching algorithm. The results of the tree pattern matchings are joined together to generate the final result for pattern graph matching.

In summary, our contributions are as follows:

- We define a *pattern graph* and give a linear time algorithm to convert multiple correlated path expressions to a pattern graph. Therefore, evaluating multiple path expressions can be transformed to the problem of matching a directed graph against an ordered tree. We call this the *graph pattern matching* (GPM) problem.
- We present rewriting rules that can be applied to a pattern graph to produce a set of equivalent pattern graphs from which the physical algebra optimizer can choose based on an appropriate cost function.
- In one of the rewriting techniques, a pattern graph is divided into connected components each of which is evaluated by a *tree pattern matching* (TPM) operator. We develop an $\Theta(nm)$ algorithm for the TPM operator, where n and m are the size of XML document and pattern tree, respectively.

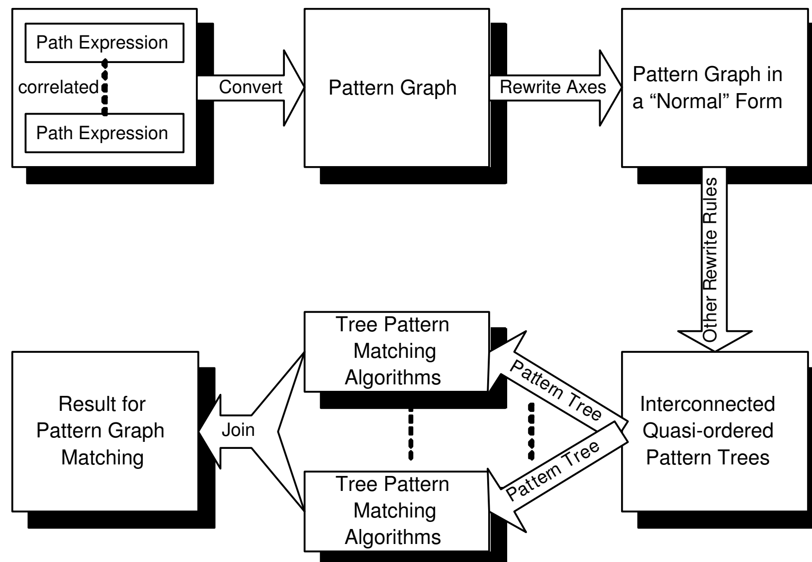

Figure 2: Overview of Optimization Steps

The rest of paper is organized as follows. In Section 2, we introduce path expressions in XML query languages and path expressions in object-oriented databases. In Section 3 we give the definitions and introduce the mapping from correlated path expressions to pattern graph. In Section 4, we present the rewriting techniques based on pattern graph. In Section 5, we use the existing algorithm combined with our own algorithm to solve the TPM problem. Finally, we introduce related works in Section 6 and conclude the paper in Section 7.

2 Background

In this section, we first introduce the syntax of path expression that is defined in the working draft of XPath 2.0 [1]. Then we discuss path expression evaluation within the context of object databases, since there are surface similarities and important differences.

2.1 Path Expressions in XPath

The syntax of path expression can be simplified as the following EBNF grammar:

Path	::=	(Root ‘/’)? Step (‘/’ Step)∗
Root	::=	‘$’ QName | ‘document’ ‘(’ StringLiteral ‘)’? | ϵ
Step	::=	Axis ‘::’ NodeTest Qualifier∗
NodeTest	::=	NameTest | KindTest
Qualifier	::=	‘[’ Exp ‘]’ | ‘=>’ NameTest
NameTest	::=	Name (‘|’ Name)∗
Name	::=	‘∗’ | ‘∗ :’? QName | QName ‘: ∗’?
Axis	::=	‘child’ | ‘parent’ | ‘descendant’ | ‘ancestor’ |
		‘following-sibling’ | ‘preceding-sibling’ | ‘self’ |
		‘descendant-or-self’ | ‘ancestor-or-self’ |
		‘following’ | ‘preceding’ | ‘attribute’ |
		‘namespace’
KindTest	::=	‘processing-instruction’ ‘(’ StringLiteral? ‘)’
		‘comment’ ‘(’ ‘)’ | ‘namespace’ ‘(’ ‘)’ | ‘node’ ‘(’ ‘)’

Each Path expression consists of a series of Steps separated by slashes (“/”). Each Step consists of an Axis, a NodeTest, and zero or more Qualifiers. The Axis can be thought of as the “direction” of this Step, representing the relationship between the previous Step and the current Step[1]. A NodeTest in the path expression filters nodes by its name or kind (e.g. comment or namespace), which are called NameTest or KindTest, respectively. Qualifiers are filters testing more complex conditions by evaluating the “qualifier expressions” enclosed in the brackets “[ ]”.

Qualifier expressions can be of any type (arithmetic, logical, for, etc), but here we restricted them to two types: i) path expressions, and ii) comparison expressions between path expression and literals (numerical and string). For example, `/bib/book[title='XML'][price<100]` is a path expression within our scope of discussion because the two qualifier expressions are comparisons between path expressions and literals; while `/bib/book/chapter[2]/title` is not, since “2” is not a path expression nor a comparison expression between path expression and literals. Another feature of the path expression that is outside the scope of this paper is the dereferencing of IDREF and IDREFS indicated by the “=>” in the Qualifier.

2.2 Path Expressions in OQL

Path expressions are not unique in XML query languages, but they also appear in object-oriented database (OODB) query languages such as OQL [14, 15]. Although they are based on different data models (ordered tree versus directed graph), path expressions in both kinds of query languages have similarities.

In OQL, path expressions are defined as a chain of objects and methods/attributes in the so-called *object composition graph* [15] (a.k.a. *aggregation graph* [16]). In the object composition graph, an object o_1 has a directed edge to another object o_2 if and only if o_1 has an attribute or method whose value or result is in the class of object o_2. For example,

[1] Following the abbreviations defined in [1], we use labels “.”, “/”, and “//” to represent self, child, and descendant-or-self axes, respectively.

$o.m_1(\varphi_1).m_2(\varphi_2).\cdots.m_n(\varphi_n)$ is a path expression, where o is an object instance, $m_i (1 \leq i \leq n)$ is an attribute or a method of either o (if $i = 1$) or $o.m_1.\cdots.m_{i-1}$ (if $i > 1$), φ_i is the argument of m_i if it is a method. This path expression can be thought of as a top-down path in the breadth-first spanning tree of the object composition graph rooted at o. This is analogous to the path expressions in XPath. However, the differences are important:

- In XPath, the XML trees on which the path expressions work (i.e., the tree representing the documents in the XML database) are known a priori, before the path expression is given. However, the OODB is based on a graph database, and the breadth-first spanning tree can be obtained only after the starting point of the path expression is fixed.
- The XML tree is always ordered, while the breadth-first spanning tree of object composition graph is always unordered. Ordering introduces differences to the query evaluation and optimization process.
- The path expressions in XPath provide a way to query XML trees in a multitude of ways including top-down, bottom-up, left-to-right, etc. by using axes. While in OODB, the object composition graph can be traversed through the reference links.
- In addition to the existence of axes, path expressions in XPath differ from path expressions in OODB in the NodeTest and qualifier expressions. In XPath path expressions, NodeTest can select nodes either by their kind or by their names (exact matching or regular expression); while in OODB path expressions, methods m_i are chosen only based on their names. In XPath path expression, qualifier brackets "[ ]" have simple and predefined semantics: the result of qualifier expressions in brackets are converted to boolean values by the predefined conversion rule, and this boolean value determines whether or not the node generated by NodeTest is filtered out. In contrast, in OODB path expressions, arguments φ_i can be any expression and the semantics of m_i varies according to the definition of types.

Path expressions in OODB are usually evaluated by joins on components along the path. The optimization techniques are therefore concentrated on how to efficiently evaluate the joins [15]. Another optimization technique is using indexed scan on the paths when an index is available. Optimization based on indexes are beyond the topic of this paper, and we refer the interested readers to [16]. Join-based evaluation of XPath path expressions is possible, and we discuss it along with an approach based on tree pattern matching (TPM), and a hybrid approach combining the two.

3 Correlated Path Expressions and Pattern Graphs

In this section, we shall first define the correlated path expressions, the subject tree and the pattern graph. Then we discuss the translation of correlated path expressions into a pattern graph.

In the rest of the paper, for simplicity, we denote subject tree and pattern graph as $\mathcal{T}$ and $\mathcal{P}$, respectively.

3.1 Definitions

Many XML queries include multiple path expressions. In most cases, these path expressions are not independent, but are somehow correlated. In this paper, we consider two simple yet widely used types of correlations for query optimization. The context in which these correlations can be used for optimization is discussed in Section 3.

Definition 1 (Correlated Path Expressions) Path expression e is said to be *correlated* to another path expression f (assuming they are bound to variables `$e` and `$f` respectively), if and only if

1. the first `Step` of e is the variable reference `$f`, i.e. the syntax of e is `$f('/'Step)*` (called *strong correlation*); or

2. they are connected by a binary operator $\theta \in \mathcal{R}$ $(e\ \theta\ f)$ that returns a boolean value. The semantics of operators in $\mathcal{R}$ are as defined in [17] (examples are the document order operator $<<$ and $>>$ or the equality operator $=$). We call this kind of correlation *weak correlation.* □

For example, path expression `$seller/userid` in Figure 1 in line 7 is correlated with the path expression `document('users.xml')//user_tuple` in line 1 since the former references the variable binding of the latter. Also, the two path expressions in line 7 are correlated to each other by the equality relation.

An XML document is usually modeled as a labeled ordered tree [18], where each node belongs to one of the seven "kinds"[2]: document, element, attribute, namespace, processing instruction, comment, and text. Each kind is associated with a set of methods defined as its access interface. This is analogous to the association of data structures and methods found in object-oriented programming languages. In our definition, subject tree is an abstract data structure for labeled ordered tree, where edges represent parent-child relationship between elements in the XML document.

Definition 2 (Subject Tree) A subject tree $\mathcal{T}$ is a rooted, ordered, node-labeled tree, which can be denoted by a 4-tuple $\mathcal{T} = \langle \Sigma, \mathcal{N}, \mathcal{E}, t \rangle$, where Σ is a finite alphabet, $\mathcal{N}$ and $\mathcal{E}$ are the sets of nodes and edges in the tree, respectively, and $t \in \mathcal{N}$ is the root of $\mathcal{T}$. A node corresponds to an element in the XML document. A node m is child of node n iff the corresponding element of m is a subelement of the corresponding element of n.

For each node $n \in \mathcal{N}$,

- the *level* of n, denoted by $level(n)$, is defined by the number of edges in the path from t to n;

- n is labeled with a character in Σ, where its label is denoted by $label(n)$;

- n is associated with a value, which is denoted by $value(n)$;

- the children of n are ordered from left to right. We call this *local order* (denoted by $\lhd$) since the ordering relationship only applies to siblings. There is also a total order among all nodes in the tree — the order obtained by preorder traversal of the tree, which we call *global order* (denoted by $\blacktriangleleft$). □

Since a subject tree is a data structure for modeling an XML document, we need to define a data structure to model path expressions. It turns out that tree is not a sufficient data structure for this purpose. Consider the path expression "`//a//b/ancestor-or-descendant::c`". This cannot be translated into a pattern tree, since the tree node `b` has two ancestors (`a` and `c`), which violates the definition of tree. Therefore, path expressions are converted to pattern graphs.

[2] This is the XPath term that roughly corresponds to a "type".

Definition 3 (Pattern Graph) A pattern graph is a labeled, directed graph, which is denoted by a 4-tuple $\mathcal{P} = \langle \Sigma, \mathcal{V}, \mathcal{A}, \mathcal{R} \rangle$, where Σ is a finite alphabet, $\mathcal{V}$ and $\mathcal{A}$ are the sets of vertices and arcs (directed edges) in the graph, respectively, and $\mathcal{R}$ is the set of binary relations[3] between vertices as defined in Definition 1.

For each vertex $v \in \mathcal{V}$:

- v is labeled with $*$ or a set of characters in Σ, where $* \notin \Sigma$.
- v is associated with a list (may be empty) of $\langle \circ, l \rangle$ tuples, where $\circ$ is a comparison operator and l is a numerical or string literal.

Each arc $(s, t) \in \mathcal{A}$ is labeled with a relation $r \in \mathcal{R}$, which indicates that (s, t) is in relation r. □

In the pattern graph, we can think of the labels for vertices and arcs as query conditions or constraints when mapping on the subject tree. Their semantics are defined in the following.

Definition 4 (GPM problem) Given a pattern graph $\mathcal{P} = \langle \Sigma, \mathcal{V}, \mathcal{A}, \mathcal{R} \rangle$ and a subject tree $\mathcal{T} = \langle \Sigma, \mathcal{N}, \mathcal{E}, t \rangle$, the GPM problem is to find all functions $\delta_i : \mathcal{V} \rightarrow \mathcal{N}$ such that:

1. For any $v \in \mathcal{V}$:
   - If $label(v) \subseteq \Sigma$, then v can be mapped to any subject tree node whose label is in $label(v)$, i.e. $label(v) \subseteq \Sigma \implies label(\delta_i(v)) \in label(v)$.
   - If $label(v) = *$, then v can be mapped to any subject tree node (we don't care about its label), i.e. $label(v) = * \implies \exists x \in \mathcal{N}.\ \delta_i(v) = x$.
   - If the associated $\langle \circ, l \rangle$ tuple list is not empty, then $value(\delta_i(v)) \circ l$ holds for every tuple in the list.

   If all the above conditions hold, $\delta_i(v)$ is called a *hit* of v.
2. For any $(s, t) \in \mathcal{A}$, if the arc is labeled with the binary relation r, then $\langle \delta_i(s), \delta_i(t) \rangle$ satisfies relation r in the subject tree.

If every vertex in $\mathcal{V}$ has a hit after the above matching process, the mapped trees in the subject tree $\delta_i(\mathcal{P})$ are called *witness trees* and form the result of GPM[4]. In this case, we say $\mathcal{P}$ is *satisfiable* on $\mathcal{T}$. □

For example, the following XPath expression can be converted to the pattern graph shown in Figure 3(b).

```
for $b in a/*,
    $a in $b/../a
    $d in $b//d,
    $c in $b/..//c,
    $e in $c/e
where $b << $a
return
    <result>
        { $d }
        { $e }
    </result>
```

[3]In this paper, we don't study ternary or higher arity relations, in which case the graph is a hypergraph.

[4]We don't consider the order of witness trees in the output here, since their order is important only in the construction process, which is outside of the scope of this paper.

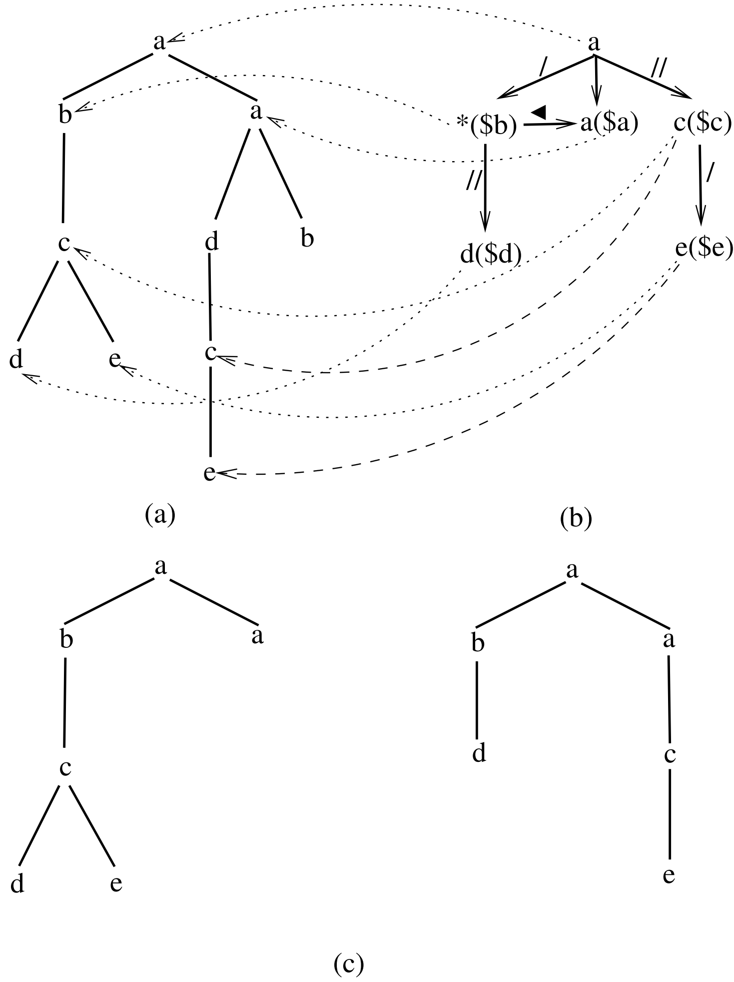

Figure 3: Example of subject tree, pattern graph, mappings, and witness trees. (a) Subject tree $\mathcal{T}$, (b) Pattern graph $\mathcal{P}$, (c) Two witness trees of $\mathcal{P}$

Assume an XML document that has been translated into the subject tree in Figure 3(a). The dotted lines from $\mathcal{V}$ in $\mathcal{P}$ to $\mathcal{N}$ in $\mathcal{T}$ represent a mapping δ_0 satisfying the definition of GPM. Note that δ_0 is not the only valid mapping. Another one differs from δ_0 only in the mappings of c and e (represented by dashed lines) in $\mathcal{V}$, where they are mapped to the c and e, respectively, in the right subtree of the root in $\mathcal{T}$. Therefore, the results of the GPM are the two witness trees corresponding to the two mappings in Figure 3(c).

3.2 Converting Correlated Path Expressions to a Pattern Graph

Based on the definition, converting correlated path expressions to a pattern graph can be done in two steps: 1) convert every path expression to a pattern graph, 2) pairwise merge multiple pattern graphs if they are correlated.

Algorithm 1 converts a path expression to a pattern graph. Based on the definition, a path expression can be divided into a sequence of Steps, each of which consists of an axis, a NodeTest and a sequence of qualifiers. The algorithm converts a Step of the path expression to a pattern graph using Algorithm 2, which, in turn, uses Algorithm 3 to convert the NodeTest that is part

Algorithm 1 ConvPath($pathExpr, r$)

Input: $pathExpr$: a path expression consisting of a list of Steps.
r is the vertex object of previous step.
Output: the vertex object corresponding to the NodeTest of the last Step.

```
1: separate pathExpr into steps S;
2: tempRoot ⇐ r;
3: for each s ∈ S in left to right order do
4:    v ⇐ ConvStep(s, tempRoot);
5:    tempRoot ⇐ v;
6: end for
7: return tempRoot;
```

Algorithm 2 ConvStep($step, pNode$)

Input: $step$: a structure consisting of axis, NodeTest and a list of qualifiers;
$pNode$: the vertex object corresponding to the NodeTest of the previous step
Output: the vertex object corresponding to the NodeTest in the current step

```
1: v ⇐ ConvNodeTest(step.NodeTest)
2: create an arc α from pNode to v;
3: label α with step.axis;
4: for each q ∈ step.qualifiers do
5:    if q is a comparison expression between path expression p and a literal l then
6:       s ⇐ ConvPath(p, v);
7:       create a tuple ⟨◦, l⟩ and associate it with s; {◦ is the comparison operator}
8:    else
9:       s ⇐ ConvPath(q, v);
10:   end if
11: end for
12: return v;
```

of that Step. Two pattern graphs resulting from two Steps are then merged together using a directed arc labeled with the axis between these two Steps.

Given two pattern graphs g and h translated from two correlated path expressions, the merge operation is straightforward: if g and h are strongly correlated and h references the variable binding of g, then we only need to create an arc from the vertex corresponding to the NodeTest of g to the vertex corresponding to the NodeTest of h (note that we didn't convert the variable reference of h in the algorithm), and label the arc with the axis connecting the variable reference and the rest of the path expression in h. If g and h are weakly correlated, we also create an arc from the vertex corresponding to the NodeTest of g to the vertex corresponding to the NodeTest of h, but label it with the binary relation between them.

For example, in the pattern graph shown in Figure 4(b), the root labeled "a" and its leftmost child, labeled "*", and the arc connecting them is a pattern graph resulting from the path expression `a/*`. The leftmost leaf, labeled with "d", is a result of converting the path expression `$b//d` without the variable reference. The arc connecting the leaf and its parent is created by merging the two strongly correlated path expressions together. The arc between the vertex labeled with "*" and the vertex labeled "a", which is the child of the root, is created by the two weakly correlated path expressions and their binary relation: "`$b << $a`".

Algorithm 3 ConvNodeTest($nodeTest$)

Input: $nodeTest$: The NodeTest of the current step
Output: The vertex object corresponding to the NodeTest.

```
1: create a vertex v;
2: if nodeTest is the wildcard "*" then
3:    labeled v with ∗;
4: else if nodeTest is a NameTest then
5:    S ⇐ ∅;
6:    for each Name separated by | do
7:       if Name is the form QName then
8:          map QName to a character c in Σ
9:          S ⇐ S ∪ {c};
10:      else if nodeTest is the form QName:* then
11:         map QName to a character c in Σ;
12:         S ⇐ S ∪ {"c : ∗"};
13:      else if nodeTest is the form QName1:QName2 then
14:         map QName1 and QName2 to c1 and c2 in Σ, respectively;
15:         S ⇐ S ∪ {"c1 : c2"};
16:      end if
17:      labeled v with S;
18:   end for
19: else if nodeTest is a KindTest then
20:    map the KindTest to its corresponding special character c; {For example, we can assign
       character @ to attribute, ∼ to namespace, # to comment, ! to PI, etc.}
21:    label v with c;
22: end if
23: return n;
```

Theorem 1 *Algorithm 1 translates a path expression to a pattern graph such that:*

1. *each NodeTest in the path expression is converted to a vertex in the pattern graph. If the NodeTest is a NameTest, the list of Name's corresponds to the label (a set of character in Σ) of the vertex.*

2. *if two vertices u and v correspond to two NodeTests in two consecutive Steps in the left-to-right order, then there is a directed arc (u, v) in the pattern graph and the label of the arc is the Axis between these two Steps.*

3. *if vertex u corresponds to a NodeTest in a Step, and u corresponds to a NodeTest in that Step's Qualifier expression, then there is a directed arc (u, v) in the pattern graph and the label of the arc is the Axis of the NodeTest in the qualifier expression. If the qualifier expression is a comparison expression between a path expression and a literal, a tuple $\langle \circ, l \rangle$ is constructed to associated with v.*

PROOF Since a path expression is a list of Steps, which consists of an Axis, a NodeTest and a list of Qualifiers. Algorithm 1 translated a path expression to a pattern graph by translating each Step into a pattern graph and then connect two pattern graphs corresponding to two consecutive Steps by a direct arc that is labeled with the Axis of the second Step. This guarantees the property 2 of the theorem.

For each Step, Algorithm 1 calls Algorithm 2 to translate a Step into a pattern graph. Algorithm 2 first calls Algorithm 3 to convert the NodeTest into a vertex in the pattern graph. If there are qualifiers, Algorithm 2 checks if the qualifier expression is a path expression or a comparison expression between a path expression and a literal. In either case, Algorithm 1 to construct another pattern graph and connect the two pattern graphs with a directed arc labeled with the Axis of the qualifier expression. If the qualifier expression is a comparison expression, Algorithm 2 also construct a tuple $\langle \circ, l \rangle$ from the comparison expression and associates the tuple with the vertex corresponding to the qualifier expression. This guarantees the property 3 of the theorem.

Algorithm 3 converts a NodeTest to a vertex in the pattern graph, with label corresponds to the Names of the NodeTest or KindTest. This guarantees the property 1 of the theorem. ∎

Theorem 2 *Algorithm 1 converts a path expression to a pattern graph in $\Theta(n)$ time, where n is the number of NodeTests.*

Proof The algorithms converts each NodeTest to a vertex in the pattern graph, and algorithm 3 runs in $\Theta(1)$ time, so the complexity of converting all NodeTests is $\Theta(n)$. Since there are exactly n axes for n NodeTests, the time for creating all edges is also $\Theta(n)$. By definition, there are at most n comparison expressions in the qualifiers, so there are $O(n)$ $\langle \circ, l \rangle$ tuples associated with vertices. Therefore, the total time for converting a path expression is $\Theta(n) + \Theta(n) + O(n) = \Theta(n)$. ∎

Careful readers might have noted that not all correlated path expressions in an XQuery expression are applicable to translating into pattern graph for evaluation without changing the semantics. For example, assume that e_1 and e_2 are two correlated path expressions where e_1 appears in a `for` clause and e_2 appears in the `where` clause. If there is a construction expression related to e_1 between e_1 and e_2, they cannot be translated into a pattern graph. Consider the following XPath expression:

```
1.    for $a in document('bib.xml')/bib/book/authors,
2.    return
3.        <author>
4.            { $a/first_name }
5.            for $b in document('articles.xml')/articles
6.            where $b/author/last_name = $a/last_name
7.            return
8.                <book> $b/title </book>
9.        </author>
```

The path expression `$a/last_name` in line 6 is correlated to the path expression `document('bib.xml')/bib/book/authors` in line 1. However, they cannot be merged together since doing so would result in a witness tree that contains a node labeled `last_name`. This is not equivalent to the semantics of the original XPath expression since the original outputs all authors' `first_name` (line 4) even if the author does not have a `last_name`. In this paper, we assume that the inputs of the conversion algorithm are correlated path expressions that can be converted to pattern graphs without changing semantics. How to identify these correlated path expressions is among our future work.

Also note that the resulting pattern graph generated by the above algorithm may not be a minimum one in the sense that there may be redundant vertices in the pattern graph. For example, the two path expressions `/bib/book[price<100]` (which is bound to variable `$b`) and `$b/price` convert to a pattern graph in which vertex labeled with "`book`" has two children, both of which are labeled "`price`". Interested readers are refereed to [19].

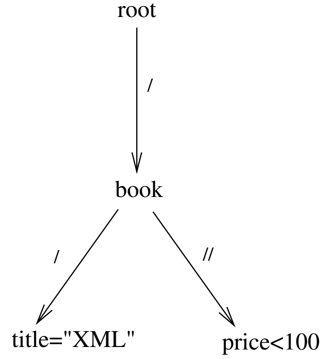

Figure 4: Pattern Graph for `/book[title=‘‘XML"][//price<100]`

4 Optimization Based on Pattern Graphs

In this section, we discuss a general evaluation strategy for path expressions based on joins and some optimization techniques based on pattern graphs transformation. The first optimization technique is a set of rewriting rules that rewrite an arc labeled by any axis to a sub-pattern graph that only contains a minimal subset of axes. The second optimization technique divides the pattern graph into connected pattern trees of "/"-arcs. Each pattern tree can be passed to a TPM operator instead of a set of joins.

4.1 A General Evaluation Strategy Based on Joins

A pattern graph $\mathcal{P}$ is a visual representation of constraints of tree nodes specified in the path expression. For example, path expression `/book[title="XML"][//price<100]` returns the books whose titles are "XML" and prices are less than 100. Its semantics can be expressed using the following conjunctive calculus expression[5]:

$$\begin{aligned}\{\ x \mid \exists y, z \quad & (\textsf{child}(root, x) \wedge label(x) = \text{``}book\text{''} \wedge \\ & \textsf{child}(x, y) \wedge label(y) = \text{``}title\text{''} \wedge value(y) = \text{``}XML\text{''} \wedge \\ & \textsf{descendant-or-self}(x, z) \wedge label(z) = \text{``}price\text{''} \wedge value(z) < 100)\}\end{aligned}$$

where *root* is the root of the current XML tree, child (descendant-or-self) is a predicate that returns true if the second argument is a child (descendant-or-self) node of the first argument in the tree. In general, any axis can be thought of as a predicate on two tree nodes. As suggested by their names, *label* and *value* are two functions that return the label and value of the tree node, respectively. Figure 4 is a pattern graph corresponding to the path expression and the conjunctive calculus expression.

This conjunctive calculus can be evaluated by converting it to an expression consisting of selection (σ) and join ($\bowtie$) operators similar to the ones in the relational algebra:

$$\sigma_{\varphi_1}(\mathcal{T}_1) \bowtie_{\textsf{child}(\mathcal{T}_1,\mathcal{T}_2)} \sigma_{\varphi_2}(\mathcal{T}_2) \bowtie_{\textsf{descendant-or-self}(\mathcal{T}_1,\mathcal{T}_3)} \sigma_{\varphi_3}(\mathcal{T}_3)$$

where all $\mathcal{T}_i$ refer to the same subject tree $\mathcal{T}$, but appropriately renamed. The three selection conditions are:

[5]Note that not all path expressions can be expressed by *conjunctive* calculus, since the latter does not allow $\vee$ and $\neg$ in the qualifier expressions.

φ_1: $label = \text{“}book\text{”}$
φ_2: $label = \text{“}title\text{”} \wedge value = \text{“}XML\text{”}$
φ_3: $label = \text{“}price\text{”} \wedge value < 100$

A possible access plan is to first evaluate the label and value functions and obtain three sequences of tree nodes x, y, and z, respectively, either by index searching or by sequential scan depending on whether there are indexes on tree node labels. Then the three sequences are “joined” together on the child and descendant-or-self predicates. One way of actually implementing these joins is to associate each tree node with a tuple that provides enough information for determining axis relationships. For example, the $\langle docID, preOrder, postOrder, level\rangle$ interval encoding [6] of subject tree nodes provides a way for judging any thirteen axis relationships between two nodes in constant time. However, when the subject tree is updated, the cost of updating the encodings is $\Theta(n)$ in the worst case, where n is the number of nodes in the subject tree. A recent improvement of this encoding scheme can achieve $\Theta(\log n)$ amortized update time and encoding length [13].

The problem of this approach is that every NodeTest is converted to a sequential scan or index lookup, and every Axis (including those in the qualifier expressions) is converted to a join on two sequences, which can seriously increase the number of joins and make it hard for the query optimizer to choose an efficient join order. For example, in the XQuery Use Cases [5], the total number of axes in the XQuery expressions[6] varies from 1 to 21, with an average of 5.68 if one counts the axes in construction expressions or 3.94 otherwise. In these axes, approximately 65.63% are child axes (“/”), 32.98% are descendant-or-self axes (“//”), and the rest are self axes (“.”). Figure 5 shows the number of “/” and “//” axes in the queries in the XQuery Use Cases.

Since almost two-thirds of the axes in path expressions are child, if nodes satisfying child relationships are stored close to each other, we can use the localization characteristics to accelerate the queries by using tree pattern matching rather than joins. The other axes that don't have localization characteristics can be evaluated by joins.

4.2 Rewriting Axes Relations

The set of axes defined in the path expression is not minimal in the sense that we can identify a subset of the axes that can express any of the eleven axes. In fact, the minimum set is not unique. For example, $\{., /, //, \lhd\}$ is one possible set of labels for the pattern graph to represent all the axes, while $\{., /, //, \blacktriangleleft\}$ is another. We choose the second set as the “minimum” set in this paper since “$\blacktriangleleft$” indicates the document order relationship, which is widely used in XPath expression evaluation. On the other hand, if a pattern graph is very large, sometimes we would like to reduce the number of arcs by combining a particular component of pattern graph into a single arc. This can be done by the reverse process of the above conversion. These conversions are provided in the form of rewriting rules in Theorem 3.

Theorem 3 (Rewriting Rules for Axes) *In the pattern graph, any arc labeled with an axis other than* attribute *and* namespace *can be converted to a subgraph whose arc labels are in the set* $\{., /, //, \blacktriangleleft\}$. *Assume that* (p, c) *is any arc and* $\lambda(p, c)$ *denotes that the axis associated with the arc is “*λ*”, depending on its label the rewriting rules are as follows:*

(a) $\mathsf{child}(p, c) \iff label(p, c) = \text{“/”}$
(b) $\mathsf{parent}(p, c) \iff label(c, p) = \text{“/”}$
(c) $\mathsf{descendant}(p, c) \iff \exists x\ label(p, x) = \text{“/”} \wedge label(x, c) = \text{“//”} \wedge label(x) = *$
(d) $\mathsf{ancestor}(p, c) \iff \exists x\ label(x, p) = \text{“/”} \wedge label(c, x) = \text{“//”} \wedge label(x) = *$

[6]The number is calculated by summing up all axes in all path expressions in an XQuery expression.

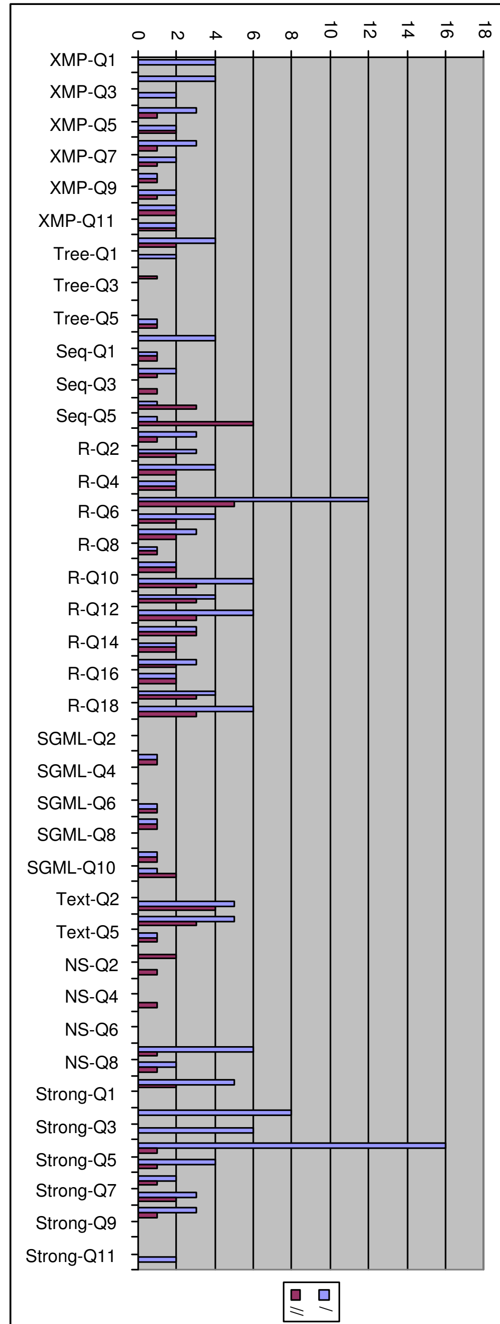

Figure 5: The numbers of axes in the queries in XQuery Uses Cases

(e) $\textsf{self}(p, c) \iff p \wedge c$
(f) $\textsf{descendant-or-self}(p, c) \iff label(p, c) = \text{“//”}$
(g) $\textsf{ancestor-or-self}(p, c) \iff label(c, p) = \text{“//”}$
(h) $\textsf{following-sibling}(p, c) \iff \exists x\ label(x, p) = \text{“/”} \wedge label(x, c) = \text{“/”} \wedge label(p, c) = \text{“} \blacktriangleleft \text{”}$
(i) $\textsf{preceding-sibling}(p, c) \iff \exists x\ label(x, p) = \text{“/”} \wedge label(x, c) = \text{“/”} \wedge label(c, p) = \text{“} \blacktriangleleft \text{”}$
(j) $\textsf{following}(p, c) \iff label(p, c) = \text{“} \blacktriangleleft \text{”}$
(k) $\textsf{preceding}(p, c) \iff label(c, p) = \text{“} \blacktriangleleft \text{”}$

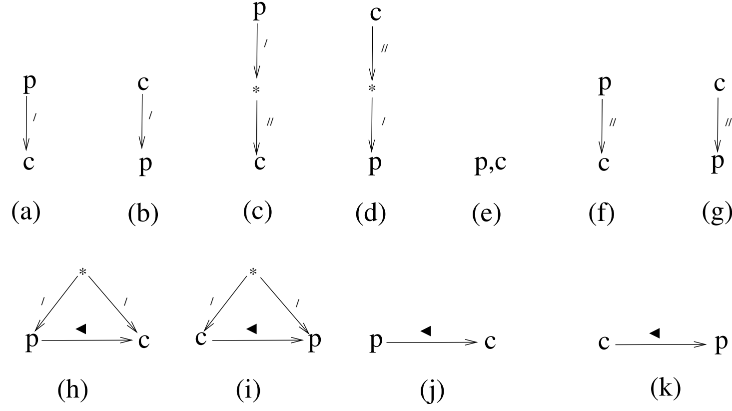

Figure 6: Converting from axes to pattern trees. (a) child(p,c) (b) parent(p,c) (c) descendant(p,c) (d) ancestor(p,c) (e) self(p,c) (f) descendant-or-self(p,c) (g) ancestor-or-self(p,c) (h) following-sibling(p,c) (i) preceding-sibling(p,c) (j) following(p,c) (k) preceding(p,c)

Their graphical representations are in Figure 6(a)–(k) in that order.

Proof The child, parent, descendant-or-self, ancestor-or-self, self, following, and preceding axes are straightforward based on the semantics of edge labels "/", "//", "." and "◀". Since ancestor is the reverse of descendant, and preceding-sibling is the reverse of following-sibling, we prove descendant and following-sibling only. For rule (c), by the semantics of descendant-or-self axis, $descendant\text{-}or\text{-}self(x, c) \equiv descendant(x, c) \vee x = c$. Therefore, the right-hand-side can be rewritten to

$$\begin{aligned} & \exists x\ child(p, x) \wedge descendant(x, c) \vee child(p, x) \wedge x = c \\ \Longleftrightarrow\ & \exists x\ descendant(p, c) \vee child(p, c) \\ \Longleftrightarrow\ & descendant(p, c) \end{aligned}$$

which is the left hand side.

Rule (h) is straightforward since the right-hand-side is exactly the semantics of following-sibling axis. ■

The purpose of rewriting axes is twofold: to convert an arc labeled by any axis to a subgraph in Figure 6 (*forward rewrite*), or vice versa, and to convert a subgraph in Figure 6 to a single arc (*backward rewrite*). Whether the query optimizer should take one direction or the other depends on the cost of the two plans. Usually the costs vary depending on the underlying physical storage structure. If the XML documents are shredded (e.g. using the interval encoding scheme [6]) and then the pieces are stored in a relational database system, backward rewrite may result in a more efficient physical plan since each arc in the pattern graph is translated into a join in the relational system [6, 7, 8], and the conversion reduces the number of the joins.

On the other hand, if the subject tree nodes are not shredded but clustered in the physical storage according to the parent-child relationship, then the forward rewrite may lead to a more efficient physical plan, since we can use the rewriting rules presented in the next subsection to divide a pattern graph into connected pattern trees. For each pattern tree, a tree pattern matching algorithm may need only one sequential scan of the data to find all witness trees. The

total cost (I/O cost plus the TPM algorithm in main memory) may be less than the cost of many sequential scans to find the nodes first and then many joins to get the final witness trees.

4.3 Partitioning Pattern Graph into Pattern Trees

Another type of rewriting rule is to partition the pattern graph into connected components such that there are no "/"-arcs connecting vertices in different components. Based on the properties of "/"-arcs, we can further transform each component such that the "/"-arcs form a tree structure, and all "//"-arcs are safely removed. Therefore, every resulting component is a tree (with optional "◀"-arcs between tree nodes) if we only consider "/"-arcs. We call this *component pattern tree* (CPT).

By rewriting each component as a CPT, we can match the CPT against the subject tree first. Then all the resulting witness trees corresponding to the CPTs are joined together according to arcs connecting them. The purpose of removing "//"-arcs is to reduce the size of the CPT and make the TPM algorithm more efficient.

Algorithm 4 DecomposePG($\mathcal{P}$)

Input: $\mathcal{P} = \langle \Sigma, \mathcal{V}, \mathcal{A}, \mathcal{R} \rangle$: the pattern graph.
Output: $\mathcal{P}$ is rewritten, or return UNSATISFIABLE if the pattern graph is unsatisfiable on any subject tree.

{INITIALIZATION}
1: $S \Leftarrow R \Leftarrow \emptyset$;
2: **for each** vertex $v \in \mathcal{V}$ **do**
3: $v.color = WHITE$;
4: $v.level = 0$;
5: **if** v's indegree of "/"-arcs is 0 **then**
6: $S \Leftarrow S \cup \{v\}$;
7: **end if**
8: **end for**
9: **if** $S = \emptyset$ **then**
10: return UNSATISFIABLE;
11: **end if**
{PARTITION THE PATTERN GRAPH INTO COMPONENTS AND COMB EACH COMPONENT INTO PATTERN TREE.}
12: $nComp \Leftarrow 0$;
13: **for each** $r \in S$ **do**
14: $R \Leftarrow R \cup Comb(\mathcal{P}, r, nComp)$;
15: $nComp \Leftarrow nComp + 1$;
16: **end for**
{REMOVE "//"-ARCS IN THE PATTERN TREES.}
17: **for each** $r \in R$ **do**
18: $RmAD(\mathcal{P}, r)$;
19: **end for**

The decomposition process is shown in Algorithm 4. The algorithm first colors each vertex as WHITE and initializes its level (the number of "/"-arcs along the path from the root) to 0. The initialization also keeps record of all vertices that have no incoming "/"-arcs and the total number of vertices whose indegree is non-zero. If no vertex has zero indegree of "/"-arcs, either

there is no "/"-arc at all or there exists a cycle of "/"-arcs. In the former case, every vertex is a component and thus a trivial CPT. For the latter case, since any "/"-arc is mapped to an edge in the subject tree, which has no cycles, the existence of a cycle in $\mathcal{P}$ implies that it is unsatisfiable on any subject tree.

The zero indegree vertices are candidates to be the roots of the resulting pattern trees. Therefore, for each of these vertices, the decomposition algorithm calls function *Comb* (shown in Algorithm 5) to explore the pattern graph using depth-first search (DFS) and manipulate the "/"-arcs to form a tree structure. The output of *Comb* is the real root of the pattern tree. After getting all of the roots of pattern trees, the decomposition algorithm calls function *RmAD* (shown in Algorithm 6) to remove "//"-arcs in the component. Both functions check the satisfiability of the pattern graph, and call function *MergePaths* (shown in Algorithm 7) to merge two paths to eliminate the cycle or redundant "//"-arcs in the "/" tree.

Algorithm 5 Comb($\mathcal{P}, r, nC$)

Input: $\mathcal{P} = \langle \Sigma, \mathcal{V}, \mathcal{A}, \mathcal{R} \rangle$: the pattern graph.
r the starting vertex (root) of the pattern tree.
nC the component #;
Output: $\mathcal{P}$ is rewritten or UNSATISFIABLE is returned.

1: $r.color \Leftarrow GREY$;
2: $r.nComp \Leftarrow nC$;
3: **for each** "/"-arc $\alpha(r, v) \in \mathcal{A}$ **do**
4: **if** $v.color = WHITE$ **then**
5: $v.level \Leftarrow r.level + 1$;
6: $v.predecessor \Leftarrow r$;
7: $Comb(\mathcal{P}, v)$;
8: **else if** $v.color = GREY$ **then**
9: return UNSATISFIABLE; {there is a directed "/" cycle}
10: **else if** $v.level \neq r.level + 1$ **then**
11: return UNSATISFIABLE; {there is a undirected "/" cycle, and v's level based on two paths are different, so unable to merge paths.}
12: **else**
13: $MergePaths(\mathcal{P}, v.predecessor, r)$;
14: **end if**
15: **end for**
16: $r.color \Leftarrow BLACK$;
17: return r;

The *MergePaths* function merges two paths in the pattern tree into one path. It takes two vertices, which have the same level, as input and test whether they are "compatible" or not. Two vertices are *compatible* if they can be mapped to the same node in the subject tree, i.e. the intersection of the labels of the two vertices is not empty, where "*" is treated as the set universe. If they are compatible, these two vertices are merged. The merging process continues recursively to the parents of the two vertices until it reaches the common ancestor of the two original two vertices.

Figure 7 shows an example of dividing a pattern graph into pattern trees. Figure 7(a) is a pattern graph consisting of "/"-arcs, "//"-arc, and "◀"-arc. The decomposition algorithm first finds the two vertices, `a` and `g`, and passes them to the function *Comb* individually. *Comb* removes all directed and undirected "/"-cycles in the left component and results in a pattern tree

Algorithm 6 RmAD($\mathcal{P}, r$)

Input: $\mathcal{P} = \langle \Sigma, \mathcal{V}, \mathcal{A}, \mathcal{R} \rangle$: the pattern graph.
r the starting vertex (root) of the pattern tree.
Output: "//"-arcs in the component are removed or UNSATISFIABLE is returned.

```
1: r.color ⇐ GREY;
2: for each "//"-arc α(r, v) ∈ A ∧ r.nComp = v.nComp do
3:   if v.color = GREY ∧ r ≠ v ∨ v.level < r.level then
4:     return UNSATISFIABLE;
5:   end if
6:   find v's ancestor u s.t. u.level = r.level
7:   MergePaths(P, r, u);
8:   remove α;
9: end for
10: if there is a "/"-arcs (r, w) s.t. w.color = BLACK then
11:   RmAD(P, w);
12: end if
13: r.color ⇐ WHITE;
```

Algorithm 7 MergePaths($\mathcal{P}, u, v$)

Require: $u.level = v.level$
Input: $\mathcal{P} = \langle \Sigma, \mathcal{V}, \mathcal{A}, \mathcal{R} \rangle$: the pattern graph.
u, v the two end vertices of "/" paths whose lengths are equal.
Output: Two paths starting from the root to u and v, respectively, are merged or UNSATISFIABLE is returned.

```
1: i ⇐ u;
2: j ⇐ v;
3: while i ≠ j do
4:   if label(i) ∩ label(j) ≠ ∅ then
5:     label(i) ⇐ label(i) ∩ label(j);
6:     append the value constraint tuple associated with j to i;
7:     remove arc (j.predecessor, j);
8:   else
9:     return UNSATISFIABLE;
10:  end if
11:  i ⇐ i.predecessor;
12:  j ⇐ j.predecessor;
13: end while
```

in the left hand side of "◀" arc shown in Figure 7(b). Then the decomposition algorithm calls function *RmAD* to remove all "//"-arcs. In the example, the only "//"-arc is removed since it is redundant with the "/"-arcs (g, h), (h, j).

Lemma 1 *Algorithm 7 merges two paths into one path if they are compatible, or return UNSATISFIABLE otherwise.*

PROOF Given two vertices i and j, we want to merge them into one vertex k such that a node in any subject tree is mapped from both i and j iff it can be mapped from k. By definition of GPM, k is mapped to a node m in the subject tree iff $label(m) \in label(k)$ and m satisfies

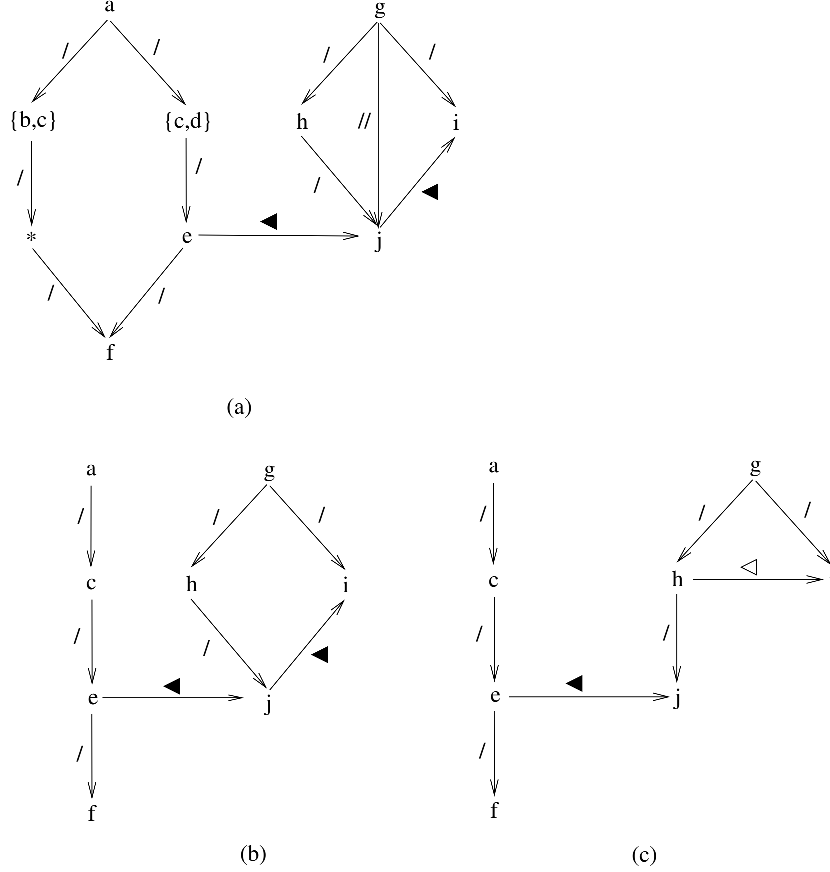

Figure 7: Decompose a Pattern Graph to Pattern Trees. (a) the original pattern graph (b) the two connected CTPs (c) the two connected quasi-ordered pattern tree

the value constraint associated with k. Similarly, m is mapped from i and j iff $label(m) \in label(i) \wedge label(m) \in label(j)$ and m satisfies the value constraints associated with both i and j. Therefore, we should label k and associate value constraints such that $label(k) = label(i) \cap label(j)$ and the value constraint is the conjunction of two constraints associated with i and j, respectively. A special case arises if one of i or j's label is "*". In this case, the result of the intersection is the label of the other vertex, i.e. "*" is treated as a set universe here. If the intersection of $label(i)$ and $label(j)$ is empty, that means that there is no node in the subject tree that can be mapped from both i and j; thus the path graph is unsatisfiable on any subject tree. ■

Lemma 2 *Algorithm 7 merges two paths of pattern graph in $O(l * |\Sigma|)$ time, where l is the length of the path from the common ancestor of u and v to u or v.*

PROOF The complexity of Algorithm 7 is straightforward: the loop is executed at most l times. In each loop, the most time-consuming process is to find the intersection of $label(i)$ and $label(j)$. Since the upper bound of $label(i)$ and $label(j)$ is $|\Sigma|$, finding the intersection can be done in $O(|\Sigma|)$ if the labels are sorted. Thus the complexity of the algorithm is $O(l * |\Sigma|)$. ■

Although the complexity result of Algorithm 7 is disappointing especially when the size of Σ is very large, the bound should be much tighter in practice since most vertex labels are expected to be a single character in Σ or "*", in which case, the complexity is only $\Theta(l)$.

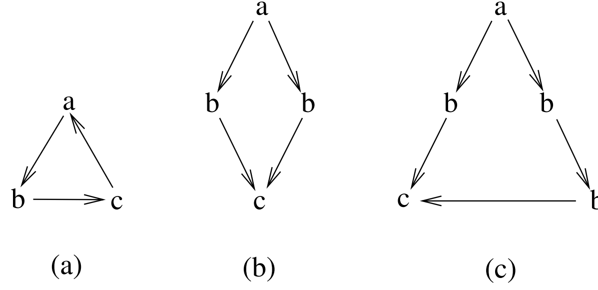

Figure 8: Breaking the "/" cycles (a) a directed cycle (b) an undirected cycle, c has the same level from two paths (c) an undirected cycle, c has different levels from two paths

Lemma 3 *Algorithm 5 divides the pattern graph into connected components, where the "/"-arcs form a tree structure in each of the connected components.*

Proof Algorithm 5 is based on depth-first search (DFS). At first, all vertices are colored WHITE. During the DFS exploration, if a vertex is first visited by a "/"-arc, it is colored GREY. When all its "/"-arcs are visited, the vertex is colored BLACK. In each loop of the algorithm, a vertex r is picked up and all its "/"-arcs (r, v) are examined. If the adjacent vertex v is WHITE, then a normal DFS call is applied to v recursively. If v's color is GREY, it means that v is one of the ancestors of r, which forms a directed cycle (illustrated in Figure 8(a)). Therefore, it is unsatisfiable on any subject tree. If v's color is BLACK, that means v has already been visited along another path. Therefore the indegree of "/"-arcs of v is greater than 1. Since any indegree of a tree node in the subject tree is 1 except the root, the only possibility for the pattern graph to be satisfiable is that the two parents of v actually map to the same node in the subject tree, thus the two parents are compatible and should be merged into one vertex. If the two levels are equal (illustrated in Figure 8(b)), then the algorithm calls function $MergePath$ to merge the two paths into one. The result of the merge is that the indegrees of "/"-arcs for v and all its ancestors are 1. If the two paths assign different levels to v (illustrated in Figure 8(c)), it is impossible that the two "/" paths lead to v can be merged together, and the algorithm returns UNSATISFIABLE. Since all the directed/undirected "/" cycles in the pattern graph can be broken, after the DFS all vertices have indegree of "/"-arcs equal to 1, which follows that the "/"-arcs form a tree structure. ■

Lemma 4 *Algorithm 5 runs in $O(m + n * |\Sigma|)$ time, where m and n are the number of "/"-arcs and vertices in the connected component, respectively.*

Proof The complexity of Algorithm 5 depends on the complexity of DFS and Algorithm 7. For each "/"-arc, the algorithm either calls $Comb$ recursively or calls $MergePath$. Since the complexity of DFS is $O(n + m)$ and there are at most n merges, the total complexity is $O(n + m) + O(n * |\Sigma|) = O(m + n * |\Sigma|)$. ■

Also since most of the labels of vertices are single character or "*", the above complexity result can be as tight as $\Theta(m + n)$.

After Algorithm 5, all vertices should be colored BLACK. To save time, we do not initialize the vertices to WHITE. Rather, in Algorithm 6, BLACK vertices indicate that they are not visited yet, while WHITE vertices mean that they are finished visiting.

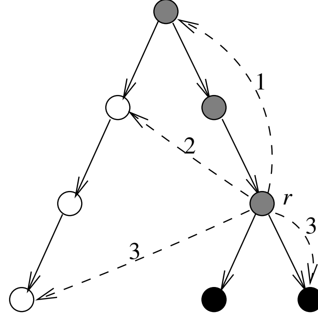

Figure 9: Remove all "//"-arcs (solid lines represent "/"-arcs and dotted lines represent "//"-arcs

Lemma 5 *Algorithm 6 removes all "//"-arcs from the a connected component.*

PROOF Algorithm 6 is also based on DFS. During the DFS when a vertex r is examined, each "//"-arc (r, v), where v is also in this connected component, is examined to see whether it is redundant. There are three possibilities (illustrated by the four dotted lines in Figure 9) for the location of v:

1. v is an ancestor of r (indicated by $v.color = GREY$ and $r \neq v$): this contradicts the constraint of (r, v) arc, which requires v is a descendant of r or r itself. So the algorithm returns UNSATISFIABLE;

2. $v.level < r.level$: this contradicts the constraint of (r, v) arc, which requires v's level is greater or equal to r's level. So the algorithm also returns UNSATISFIABLE;

3. $v.level \geq r.level$: either v is a descendant or self of v, or r and v are in different paths from the root, the necessary condition for the pattern graph to be satisfiable is that the two paths from the root to r and v can be merged together. Thus the algorithm calls the $MergePaths$ function to merge the two paths into one, and then remove the "//"-arc.

Since all "//"-arcs are removed in all cases, the resulting component is a pattern tree consisting of only "/"-arcs and possibly "◀"-arcs. ■

Lemma 6 *Algorithm 6 runs in $O(m+n*|\Sigma|)$ time, where m is the number of "/" and "//"-arcs, and n is the number of vertices in the pattern tree.*

PROOF The complexity analysis is similar to the analysis of Algorithm 5. The complexity for DFS is $O(m+n)$ and there are at most n vertices merges, so the the total complexity is $O(m+n*|\Sigma|)$. ■

Theorem 4 *Algorithm 4 decomposes a pattern graph into connected CPT's.*

PROOF This theorem follows directed from the correctness analysis of Algorithm 5, Algorithm 6 and Algorithm 7. ■

Theorem 5 *Algorithm 4 decomposes a pattern graph into pattern trees in $O(m + n * |\Sigma|)$ time, where n and m are the number of vertices and arc, respectively, in the pattern graph.*

PROOF The complexity of this algorithm also follows directed from the complexity analysis of Algorithm 5 and Algorithm 6: $O(m + n * |\Sigma|)$. ■

5 Tree Pattern Matching Algorithms

Once the pattern graph is decomposed into connected CPT's, each CPT can be evaluated by a TPM operator (τ) that takes a pattern tree as input and produces a sequence of witness trees. The motivation for using τ operator is that vertices connected by "/"-arcs are mapped to parent and child nodes in the subject tree. If all adjacent nodes in the subject tree are clustered in the physical storage, a "sequential scan" may be more efficient than a join on two sequence of nodes based on child relationship.

The logical τ operator can be implemented as a sequential scan, or it can be converted to a physical algebraic expression. By sequential scan, we mean a TPM algorithm that traverses the subject tree in preorder and outputs the witness tree during the traversal. In the second case, the pattern tree is broken up into many subtrees (could be a single vertex), such that each subtree can be evaluated by a sequential scan or by looking up the path indexes if available. The results of the sub-queries are joined together based on the child relationships.

Since a pattern tree obtained by the decomposition algorithm consists of "◀"-arcs as well as "/"-arcs, and "◀" relation does not demonstrates locality as by "/" relation. We need to convert "◀"-arcs into "◁"-arcs, which are the relation between siblings, thus more "local" than "◀" relations.

The conversion algorithm is shown in Algorithm 8, and the result of localizing Figure 7(b) is shown in Figure 7(c).

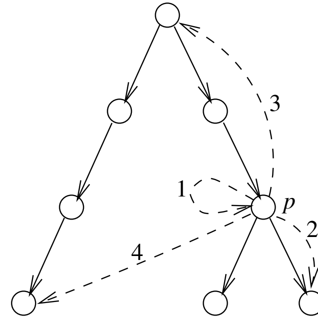

Figure 10: Replace "◀"-arcs with "◁"-arcs (solid lines represent "/"-arcs and dotted lines represent ""-arcs)

Theorem 6 *Algorithm 8 removes all "◀"-arcs.*

PROOF For every "◀"-arc (p, q), there are four possibilities (illustrated in Figure 10:

1. p and q are the same vertex: $p \blacktriangleleft p$ contradicts the irreflexive property of "◀" relation, so the pattern tree is unsatisfiable on any subject tree.

2. p is an ancestor of q: p being an ancestor of q implies that $p \blacktriangleleft q$. So the relation $p \blacktriangleleft q$ is redundant and the "◀"-arc (p, q) can be safely removed without adding any "◁"-arcs.

3. p is an descendant of q: p being q's descendant implies that $q \blacktriangleleft p$, which contradicts $p \blacktriangleleft q$ since the ◀ relation is anti-symmetric.

4. p and q are on different paths from the root: the fact that p and q are on different paths implies that all ancestors of p and q up to (but not including) their common ancestor are

Algorithm 8 Localize(t)

Input: t: the pattern tree.
Output: all $\blacktriangleleft$ arcs are removed and $\lhd$ arcs are added

```
1: for each ◀ arc (p, q) s.t. p.nComp = q.nComp do
2:   if p = q then
3:     return UNSATISFIABLE;
4:   end if
5:   i ⇐ p,   j ⇐ q;
6:   if i.level > j.level then
7:     i ⇐ i's ancestor s.t. i.level = j.level;
8:     if i = j then
9:       return UNSATISFIABLE; {p is a descendant of q ⟹ q ◀ p, contradiction.}
10:    end if
11:  else
12:    j ⇐ j's ancestor s.t. i.level = j.level;
13:    if i = j then
14:      remove the "◀"-arc (p, q);
15:      continue the loop from beginning; {p is an ancestor of q, just remove the "◀"-arc, no
         need to add a "◁"-arc.}
16:    end if
17:  end if
18:  while i.predecessor ≠ j.predecessor do
19:    i ⇐ i.predecessor;
20:    j ⇐ j.predecessor;
21:  end while
22:  add a "◁"-arc (i, j) to t;
23:  remove the "◀"-arc (p, q);
24: end for
```

in the $\blacktriangleleft$ relation. That is, if p' and q' are parents of p and q, respectively, and they are different vertices, $p' \blacktriangleleft q'$ holds. The same relation holds for the parents of p' and q', and so on. In particular, when p' and q' are the ancestors of p and q, respectively, and p' and q' are siblings, $p' \lhd q'$ holds. On the other hand, if we know $p' \lhd q'$, we can infer $p \blacktriangleleft q$ as well. Therefore, we can remove the "$\blacktriangleleft$"-arc and add the "$\lhd$"-arc (p', q').

Since the above four cases are all the possibilities of relationships of p and q, and in all cases, "$\blacktriangleleft$"-arcs can be replaced by "$\lhd$"-arcs or return UNSATISFIABLE, the resulting connected component has no "$\blacktriangleleft$"-arcs. ∎

Theorem 7 *Algorithm 8 removes all "$\blacktriangleleft$"-arcs in $O(l * m_\blacktriangleleft)$ time, where $m_\blacktriangleleft$ and l are the number of "$\blacktriangleleft$"-arcs and maximum depth of pattern tree, respectively.*

Proof The complexity of Algorithm 8 is straightforward: for each "$\blacktriangleleft$"-arc, we need at most l upward look-up in order to find p' and q'. So the complexity is $O(l * m_\blacktriangleleft)$. ∎

The output of Algorithm 8 is a CPT consisting of only "/" and "$\lhd$"-arcs. Since the $\lhd$ relation is a quasi-order relation satisfying irreflexive and transitive properties[7], we call the output of

[7] Notice that irreflexivity and transitivity also imply anti-symmetry. We take this definition of quasi-order from [20]. Some other textbooks define quasi-order as a relationship satisfying reflective and transitive properties.

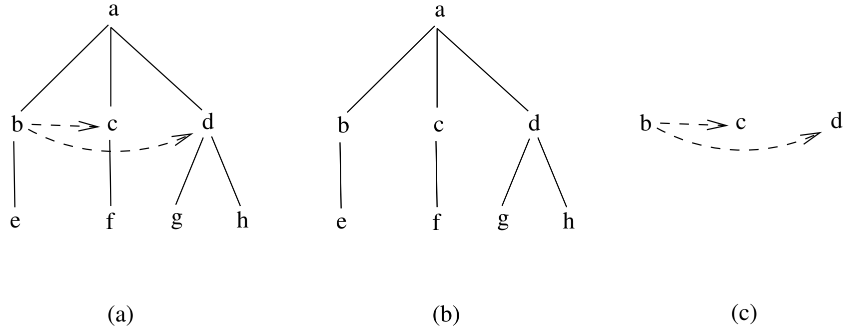

Figure 11: Converting a quasi-ordered pattern tree to an unordered pattern tree and a DAG (solid lines are "/"-arcs and dashed-directed lines are "$\lhd$"-arcs

Algorithm 8 *quasi-ordered pattern tree*. The problem of matching a quasi-ordered pattern tree with subject trees is called a *quasi-ordered pattern matching* (QTPM) problem.

A quasi-ordered pattern tree can be divided into an unordered pattern tree (UTPM), where children of a vertex are unordered, and a set of directed acyclic graphs (DAG) consisting of only "$\lhd$"-arcs. For example, the quasi-ordered pattern tree shown in Figure 11(a) can be divided into an unordered pattern tree and a DAG shown in Figure 11(b) and (c), respectively.

There is a significant body of literature addressing UTPM. We shall briefly discuss these in Section 6. Since our TPM operator is at a logical level, any of the TPM implementation can be used. The choice depends on an appropriate cost function. Physical optimization is outside the scope of this paper and is the subject of future work.

After applying the UTPM algorithm to the subject tree, the result can be further filtered out by a DAG pattern matching (DAGPM) operator. DAG pattern matching takes a sequence of lists (siblings in the witness trees) and checks whether it satisfies the constraints specified by the DAG. The algorithm is shown in Algorithm 9.

Theorem 8 *Algorithm 9 matches a DAG with a totally order list.*

PROOF A DAG is successfully matched with a list if there exists a mapping f from the nodes of the DAG to the nodes of the list such that whenever there is a directed edge from x to y in the DAG, $f(x)$ always precedes $f(y)$ in the list. Based on this property, the algorithm scans the list from left to right, tries to match each item in the list to the nodes in the pattern tree with zero indegree, and deletes the matched node in pattern tree before moving to the next item in the list. A node in the pattern tree with non-zero indegree means that there is a preceding node in the pattern tree that failed to match with the list so far because all matched nodes are deleted. So any zero-indegree node is free to match to the current item in the list. ∎

Theorem 9 *Algorithm 9 matches a DAG with a totally ordered list in $O(mn)$ time.*

PROOF There are at most n items in the list and m nodes in the DAG. For each item during the scan, the algorithm need to look up at most m nodes in the worst case. Assuming that node matching is done in $O(1)$, the total complexity is $O(mn)$. ∎

Algorithm 9 Algorithm DAGPM(dag,lst)

Input: dag: the DAG of pattern tree P;
lst: the totally ordered list of siblings in the subject tree;
Output: TRUE if dag matches lst, FALSE otherwise;

```
1: if dag = null then
2:   return TRUE;
3: else
4:   for each n ∈ lst in left-to-right order do
5:     for each d ∈ dag do
6:       if d.indegree = 0 ∧ NodeMatch(n, d) then
7:         remove d and all its incident arcs from dag;
8:       end if
9:     end for
10:  end for
11:  if dag.size = 0 then
12:    return TRUE;
13:  else
14:    return FALSE;
15:  end if
16: end if
```

6 Related Work

Managing ordered hierarchical data (particularly XML documents) has attracted significant research and development attention [21, 22, 23]. However, on the query optimization side, there is no agreement on an algebra for XML query languages so far, and it still needs to be clarified whether a "native" XML storage is needed, or XML documents should be shredded and stored in a relational database system. Many methods have been proposed to use various labeling schemes [6, 12, 13] to encode subject tree nodes such that they can be stored in a relational database system and still be able to answer structural queries (whether two nodes satisfy the parent-child/ancestor-descendant/preceding-following-sibling, etc. relationships). However, there is a lack of mechanisms to generate equivalent access plans for different physical storage managers (for example, a "native" XML storage system that clusters subject tree nodes according to the parent-child relationship). One of the purposes of this paper is to provide rewriting rules that can be used in this process.

Another purpose of this paper is to define a subset of path expression that is widely used in practice, and exploit the correlations among multiple of such path expressions in query optimization. Previous research has proposed using pattern trees (or twigs) to capture the query conditions specified by the path expressions [6, 7, 8, 9]. However none of these works takes into consideration multiple path expressions, nor do they give an algorithm to convert a set of path expressions to a pattern tree. Recent research [10] provides the semantics for the complete path expression in terms of two relations: "firstchild" and "nextsibling". However, there is still lack of optimization techniques based on the semantics.

In this paper, we provide optimization techniques based on pattern graph. One of the techniques is to reduce the complexity of pattern graph by transforming it to connected pattern trees. This process is similar to the process of minimizing pattern trees in [19]. However, the latter reduces the size of pattern trees by removing redundant nodes. In our paper, we remove

redundant arcs in the pattern graph to produce pattern trees. We can then use the minimization technique provided in [19] on the results of our pattern trees before giving them to the TPM operators.

The last purpose of this paper is to provide algorithms for QTPM problem. The solution combines the existing UTPM algorithm and our DAGPM algorithm. Depending on the ordering of pattern tree, the tree pattern matching problem is categorized into ordered tree pattern matching (OTPM) problem, unordered tree pattern matching (UTMP) problem, and quasi-ordered tree pattern matching (QTPM) problem.

OTPM problem can be solved very efficiently—a naive algorithm, which checks each node in the subject tree for occurrence of the pattern tree, only takes $O(nm)$ time, where n and m are the numbers of nodes in the subject tree and pattern tree, respectively. More efficient OTPM algorithms powered by suffix trees or other data structures can be as efficient as $O(n\sqrt{m}\ \text{polylog}\ m)$ [24, 25].

To solve UTPM problem, there are basically two approaches. One is combinatorial, relying only on the topological structure of subject trees. The other is more "database oriented" in that the subject tree is treated as a relation of 4-tuples, which is the interval encoding of subject tree nodes. Relationships between two nodes are determined by querying on the interval encoding relation.

6.1 A Combinatorial Approach to UTPM

In this approach, developed by Shasha et. al. [9] for approximate tree pattern searching, the pattern tree is an unordered tree. In this case, if the path expression specifies sibling ordering (e.g. using following-sibling axis), additional operators are required to check for the order among siblings in the witness trees.

The algorithm converts the tree pattern matching problem to the string pattern matching problem by separating the pattern and subject trees into paths (from root the leaves) so that the trees can be considered as sets of pattern strings P and subject strings S, respectively. A subtree rooted at r in the subject tree is said to be matched with the pattern tree if and only if all pattern strings match some subject strings starting from r. It is reasonably efficient when all the relationships in the pattern tree are parent-child relationships. However, if there are ancestor-descendant relationships, one has to divide the pattern tree into subtrees by deleting ancestor-descendant edges, so that every subtree does not contain ancestor-descendant relationships. Then the subtrees are matched against the subject tree individually, and their witness trees are joined together (based on the deleted ancestor-descendant relationships) to get the final witness tree. The worst case is if all relationships in the pattern tree are ancestor-descendant relationships. Its complexity largely depends on what join algorithm is used to join witness subtrees together. If no appropriate labeling techniques are used it could result in exponential complexity.

6.2 A Relational Interval Encoding Approach to UTPM

The second approach tackles the problem from a rather "relational database" perspective[6, 7, 8]. The basic idea is to encode each node in the subject tree as a 4-tuple $\langle d,\ p,\ q,\ l\rangle$, where d is a unique ID for document, p and q are the orders of this node in the preorder and postorder traversal respectively, and l is the number of edges in the path starting from the root to this node. Since the $\langle$*preorder, postorder*$\rangle$ pair represents the position interval of opening tag and closing tag of the element, the encoding scheme is called *interval encoding*. After all nodes are encoded, the subject tree can be thought of as a relation of 4-tuples. In this way, one can decide

any relationship denoted by the axes[8] by posing a query on the relation. The conditions for querying tree nodes that are related by axes to a given tree node are listed in Table 1. If using a multi-way merge join, the algorithm can be implemented very efficiently. However this approach depends on the static interval encoding scheme. When the XML tree structure is updated, the whole relation has to be updated as well.

Axes	**Logical formulae on interval encoding $\langle d, p, q, l\rangle$, assuming $x.d = y.d$**
child	$(x.p < y.p) \wedge (x.q > y.q) \wedge (x.l = y.l - 1)$
descendant	$(x.p < y.p) \wedge (x.q > y.q)$
parent	$(x.p > y.p) \wedge (x.q < y.q) \wedge (x.l = y.l + 1)$
self	$x.p = y.p$
descendant-or-self	$(x.p \leq y.p) \wedge (x.q \geq y.q)$
ancestor	$(x.p > y.p) \wedge (x.q < y.q)$
following-sibling	$(x.p < y.p) \wedge (x.q < y.q) \wedge (x.l = y.l)$
following	$(x.p < y.p) \wedge (x.q < y.q)$
preceding-sibling	$(x.p > y.p) \wedge (x.q > y.q) \wedge (x.l = y.l)$
preceding	$(x.p > y.p) \wedge (x.q > y.q)$
ancestor-or-self	$(x.p \geq y.p) \wedge (x.q \leq y.q)$

Table 1: Converting axes to logical formulae on interval encoding. x is a given subject tree node, and y is the result of axis::x, $\langle d, p, q, l\rangle$ is the interval encoding.

7 Conclusion and Future Work

In this paper, we have focused on the correlations among multiple path expressions in XML query expressions, and optimizing them by exploiting these correlations. The process converts correlated path expressions in a pattern graph, which captures all the constraints in the path expressions. We proposed rewriting rules that transform the pattern graph into a set of equivalent pattern graphs and decompose the pattern graph into connected pattern trees. Each of these pattern graphs can be thought of as a logical level algebraic expression. Given an appropriate cost function, an XML query optimizer would be able to choose an optimal one from those access plans depending on the physical level information.

Our future work includes the following:

- Implementing various TPM algorithms and comparing them with structural joins [7, 8] in different situations.
- Finding appropriate cost functions for different physical storage structures (relational or native XML) to approximate the cost of each pattern graph we generated.
- Generalizing the pattern graph or defining other operators that combines with GPM to capture the complete path expression defined by W3C.
- Defining other algebraic operators and algorithms that capture the complete XQuery expression.

[8] The axes attribute and namespace cannot be determined by interval encoding.

References

[1] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Siméon, "XML Path Language (XPath) 2.0." Available at http://www.w3.org/TR/xpath20/.

[2] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu, "XQuery 1.0: An XML Query Language." Available at http://www.w3.org/TR/xquery/.

[3] J. Clark, "XSL Transformations (XSLT) Version 1.0." Available at http://www.w3.org/TR/xslt.

[4] S. DeRose, R. D. Jr., P. Grosso, E. Maler, J. Marsh, and N. Walsh, "XML Pointer Language (XPointer)." Available at http://www.w3.org/TR/xptr/.

[5] D. Chamberlin, P. Fankhauser, M. Marchiori, and J. Robie, "XML Query Use Cases." Available at http://www.w3.org/TR/xmlquery-use-cases.

[6] C. Zhang, J. Naughton, D. Dewitt, Q. Luo, and G. Lohman, "On Supporting Containment Queries in Relational Database Management Systems," in *Proc. 20th ACM Int. Conf. on Management of Data*, pp. 425–436, 2001.

[7] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu, "Structural Joins: A Primitive for Efficient XML Query Pattern Matching," in *Proc. ICDE*, 2002.

[8] N. Bruno, N. Koudas, and D. Srivastava, "Holistic Twig Joins: Optimal XML Pattern Matching," in *Proc. 21st ACM Int. Conf. on Management of Data*, pp. 310–322, 2002.

[9] D. Shasha, J. T. L. Wang, and R. Giugno, "Algorthmics and Applications of Tree and Graph Searching," in *Proc. 21st ACM Symp. on Principles of Database Systems*, pp. 39–53, 2002.

[10] G. Gottlob, C. Koch, and R. Pichler, "Efficient Algorithms for Processing XPath Queries," in *Proc. 28th VLDB Conf.*, 2002.

[11] P. Fankhauser, M. Fernandez, A. Malhotra, M. Rys, J. Simeon, and P. Wadler, "XQuery 1.0 Formal Semantics." Available at http://www.w3.org/TR/query-semantics/.

[12] E. Cohen, H. Kaplan, and T. Milo, "Labeling Dynamic XML Trees," in *Proc. 21st ACM Symp. on Principles of Database Systems*, pp. 271–281, 2002.

[13] Y. Chen, G. Mihaila, S. Padmanabhan, and R. Bordawekar, "Labeling Your XML." preliminary version presented at CASCON'02, October 2002.

[14] J. D. Ullman and F. Widom, *A First Course in Database Systems*. Prentice Hall, 1998.

[15] W. K. (ed.), *Modern Database Management—Object-Oriented and Multidatabase Technologies*, pp. 146–174. Addison-Wesley/ACM Press, 1994.

[16] E. Bertino, B. C. OOI, R. Sacks-Davis, K.-L. Tan, J. Zobel, B. Shidlovsky, and B. Catania, *Indexing Techniques for Advanced Database Systems*. Kluwer Academic Publishers, 1997.

[17] A. Malhotra, J. Melton, J. Robie, and N. Walsh, "XQuery 1.0 and XPath 2.0 Functions and Operators." Available at http://www.w3.org/TR/xquery-operators/.

[18] M. Fernandez, J. Marsh, and M. Nagy, "XQuery 1.0 and XPath 2.0 Data Model." Available at http://www.w3.org/TR/query-datamodel/.

[19] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava, "Minimization of Tree Pattern Queries," in *Proc. ACM Int. Conf. on Management of Data*, pp. 497–508, 2001.

[20] D. F. Stanat and D. F. McAllister, *Discrete Mathematics in Computer Science.* Prentice-Hall, 1977.

[21] V. Vianu, "A Web Odyssey: from Codd to XML," in *Proc. 20th ACM Symp. on Principles of Database Systems*, pp. 1–15, 2001.

[22] F. Neven, "Automata Theory for XML Researchers," *ACM SIGMOD Record*, vol. 31, no. 3, 2002.

[23] R. Bourret, "XML Database Product." Available at http://www.rpbourret.com/xml/XMLDatabaseProds.htm.

[24] C. M. Hoffmann and M. J. O'Donell, "Pattern Matching in Trees," *J. ACM*, vol. 29, no. 1, pp. 68–95, 1982.

[25] M. Dubiner, Z. Gallil, and E. Magen, "Faster Tree Pattern Matching," *J. ACM*, vol. 41, no. 2, pp. 205–213, 1994.