

A Study of a Positive Fragment of Path Queries: Expressiveness, Normal Form, and Minimization

Yuqing Wu¹, Dirk Van Gucht¹, Marc Gyssens², and Jan Paredaens³

¹ Indiana University, USA

² Hasselt University & Transnational University of Limburg, Belgium

³ University of Antwerp, Belgium

Abstract. We study the expressiveness of a positive fragment of path queries, denoted Path^+ , on node-labeled trees documents. The expressiveness of Path^+ is studied from two angles. First, we establish that Path^+ is equivalent in expressive power to a particular sub-fragment as well as to the class of tree queries, a sub-class of the first-order conjunctive queries defined over label, parent-child, and child-parent predicates. The translation algorithm from tree queries to Path^+ yields a normal form for Path^+ queries. Using this normal form, we can decompose a Path^+ query into sub-queries that can be expressed in a very small sub-fragment of Path^+ for which efficient evaluation strategies are available. Second, we characterize the expressiveness of Path^+ in terms of its ability to resolve nodes in a document. This result is used to show that each tree query can be translated to a unique, equivalent, and minimal tree query. The combination of these results yields an effective strategy to evaluate a large class of path queries on documents.

1 Introduction

XQuery [5] is a language to express queries on XML documents (i.e., node-labeled trees). In this paper, we study the expressiveness of an algebraic path query language, denoted Path^+ , which is equivalent to a sub-language of XQuery, and wherein each query associates with each document a binary relation on its nodes. Each pair (m, n) in such a relation can be interpreted as the unique, shortest path from m to n in the queried document. Hence, whenever we talk in the paper about a path in a document, we represent it by the pair of its start- and end-node.

Consider the XQuery query on the right. We can express such queries in an algebraic path query language which we denote as the Path^+ algebra. The Path^+ algebra allows \emptyset formation, label examination, parent/child navigation, composition, first and second projections, and intersection. More precisely, the expressions of Path^+ are

```
for $i in doc(...)//a/b
  for $j in $i/c/*/d[e]
    for $k in $j/*/f
  return ($i, $k)
intersect
for $i in doc(...)//a/b
  for $j in $i/c/a/d
    for $k in $j/c/f
  return ($i, $k)
```

$$E ::= \emptyset \mid \varepsilon \mid \hat{\ell} \mid \downarrow \mid \uparrow \mid E; E \mid \Pi_1(E) \mid \Pi_2(E) \mid E \cap E$$

where the primitives \emptyset , ε , $\hat{\ell}$, \downarrow , \uparrow respectively return the empty set of path, the paths of length 0, the labeled paths of length 0, the parent-child paths, and the

child-parent paths, and the operators $;$, Π_1 , Π_2 , and \cap denote composition, first projection, second projection, and intersection of sets of paths. Path^+ is fully capable of expressing the XQuery query above in an algebraic form as

$$\Pi_2(\hat{a}; \downarrow; \hat{b}; \downarrow; \hat{c}; \downarrow; \downarrow; \hat{d}; \Pi_1(\downarrow; \hat{e}); \downarrow; \downarrow; \hat{f} \cap \Pi_2(\hat{a}; \downarrow; \hat{b}; \downarrow; \hat{c}; \downarrow; \hat{a}; \downarrow; \hat{d}; \Pi_1(\downarrow; \hat{e}); \downarrow; \hat{c}; \downarrow; \hat{f}$$

XPath is a language for navigation in XML documents [6] and is always evaluated in the node-set semantics. Researchers have introduced clean algebraic and logical abstractions in order to study this language formally. Literature on the formal aspects of XPath has become very extensive, which involves full XPath as well as its fragments [8, 3, 14, 13, 10]. Research on XPath and its sub-languages has been focusing on the expressiveness [10] and the efficient evaluation of these languages [8, 12]. Tree queries, also called pattern trees, are also natural to XML. They have been studied ever since XML and query languages on XML were introduced. Such studies cover areas from the minimization of tree queries [2, 15] to the efficient evaluation of pattern trees [1, 11, 4].

However, XQuery, with its FLWR statement and nested variable bindings, is capable of combining the results of multiple XPath queries. This language feature requires that the path expressions in XQuery be evaluated in the path semantics. In this paper, we study the expressiveness of Path^+ from two angles.

(1) We establish that Path^+ is equivalent in expressive power to a particular sub-fragment of this language as well as to the class of tree queries, a sub-class of the first-order conjunctive queries defined over label, parent-child, and child-parent predicates. The translation algorithm from tree queries to Path^+ expressions yields a normal form for Path^+ expressions. Using this normal form, we can decompose a Path^+ query into sub-queries that can be expressed in a very small sub-fragment of Path^+ for which efficient evaluation strategies are available.

(2) We characterize the expressiveness of Path^+ in terms of its ability to resolve pairs of nodes in a document. We show pairs of nodes cannot be resolved if and only if the paths from the root of the documents to these nodes have equal length and corresponding nodes on these paths are bisimilar. This result is then used to show that each tree query can be translated to a unique, equivalent, and minimal tree query.

We conclude the paper by showing that Path^+ queries can be regarded as the canonical building blocks for more general path queries, such as those involving union, set difference, ancestor, and descendant operations. As such, Path^+ can be viewed to path queries, as SPJ queries are viewed to relational algebra queries.

2 Preliminaries

In this section, we give the definition of documents, a positive fragment of path queries, and the query language of tree queries.⁴

Definition 1. *A document D is a labeled tree (V, Ed, λ) , with V the set of nodes, $Ed \subseteq V \times V$ the set of edges, and $\lambda : V \rightarrow \mathcal{L}$ a node-labeling function.*

⁴ Throughout the paper, we assume an infinitely enumerable set \mathcal{L} of labels.

For two arbitrary nodes m and n in a document D , there is a unique, shortest path from m to n if we ignore the orientation of the edges. The unique node on this path that is an ancestor of both m and n will henceforth be denoted $\text{top}(m, n)$.

Example 1. Figure 1 shows an example of a document that will be used throughout the paper. Notice that, in this document, $\text{top}(n_8, n_{12}) = n_4$.

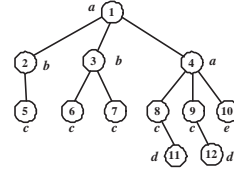


Fig. 1. An Example Document

2.1 The Positive Path Algebra

Here we give the formal definition of the positive path algebra, denoted Path^+ , and its semantics.

Definition 2. Path^+ is an algebra which consists of the primitives \emptyset , ε , $\hat{\ell}$ ($\ell \in \mathcal{L}$), \downarrow , and \uparrow , together with the operations composition ($E_1; E_2$), first projection ($\Pi_1(E)$), second projection ($\Pi_2(E)$) and intersection ($E_1 \cap E_2$). (E , E_1 , and E_2 represent Path^+ expressions.)

Given a document $D = (V, Ed, \lambda)$, the semantics of a Path^+ expression is a binary relation over V , defined on the right. By restricting the operators allowed in expressions, several sub-algebras of Path^+ can be defined. The following is of special interest to us: $\text{Path}^+(\Pi_1, \Pi_2)$ is the sub-algebra of

$\begin{aligned} \emptyset(D) &= \emptyset; \\ \varepsilon(D) &= \{(n, n) \mid n \in V\}; \\ \hat{\ell}(D) &= \{(n, n) \mid n \in V \text{ and } \lambda(n) = \ell\}; \\ \downarrow(D) &= Ed; \\ \uparrow(D) &= Ed^{-1}; \\ E_1; E_2(D) &= \pi_{1,4}(\sigma_{2=3}(E_1(D) \times E_2(D))); \\ \Pi_1(E)(D) &= \{(n, n) \mid \exists m : (n, m) \in E(D)\}; \\ \Pi_2(E)(D) &= \{(n, n) \mid \exists m : (m, n) \in E(D)\}; \\ E_1 \cap E_2(D) &= E_1(D) \cap E_2(D); \end{aligned}$
--

Path^+ where, besides the primitives and the composition operation, only the first and second projections are allowed. In addition, we will consider the algebra $\text{DPath}^+(\Pi_1)$, where, besides the primitives \emptyset , ε , $\hat{\ell}$, \downarrow , and the composition operations, only the first projection is allowed. Thus, in $\text{DPath}^+(\Pi_1)$ expressions, the primitive \uparrow and the second projection are not allowed.

Example 2. The following is an example of a Path^+ expression:

$$\Pi_1(\downarrow); \Pi_2(\hat{d}; \uparrow; \hat{c}); \Pi_2(\hat{a}; \downarrow; \hat{c}); \uparrow; \Pi_2(\Pi_1((\downarrow; \hat{a}; \downarrow) \cap (\downarrow; \downarrow; \hat{c})); \downarrow); \downarrow; \hat{c}; \Pi_1(\hat{c}; \downarrow; \hat{d}); \downarrow.$$

The semantics of this expression given the document in Figure 1 is the following set of pairs of nodes of that document: $\{(n_8, n_{11}), (n_8, n_{12})\}$. The above expression is equivalent to the much simpler $\text{Path}^+(\Pi_1, \Pi_2)$ expression

$$\Pi_1(\downarrow; \hat{d}); \hat{c}; \uparrow; \Pi_2(\downarrow); \hat{a}; \downarrow; \Pi_1(\downarrow; \hat{d}); \hat{c}; \downarrow.$$

Note that the sub-expressions $\Pi_1(\downarrow; \hat{d})$ and $\hat{a}; \downarrow; \Pi_1(\downarrow; \hat{d}); \hat{c}; \downarrow$ are in $\text{DPath}^+(\Pi_1)$.

2.2 Tree Queries

Here we define the tree query language, denoted \mathbf{T} , and its semantics.

Definition 3. A tree query is a 3-tuple (T, s, d) , with T a labeled tree, and s and d nodes of T , called the source and destination nodes. The nodes of T are either labeled with a symbol of \mathcal{L} or with a wildcard denoted “*”, which is assumed not to be in \mathcal{L} . To the set of all tree queries, we add \emptyset . The resulting set of expressions is denoted \mathbf{T} .

Two symbols of $\mathcal{L} \cup \{*\}$ are called *compatible* if they are either equal or one of them is a wildcard. For two compatible symbols ℓ_1 and ℓ_2 , we define $\ell_1 + \ell_2$ to be ℓ_1 if ℓ_1 is not a wildcard, and ℓ_2 otherwise. Let $P = ((V', Ed', \lambda'), s, d)$ be a tree query, and let $D = (V, Ed, \lambda)$ be a document. A *containment mapping* of P in D is a mapping $h : V' \rightarrow V$ such that

1. $\forall m', n' \in V' ((m', n') \in Ed' \rightarrow (h(m'), h(n')) \in Ed)$; and
2. $\forall m' \in V' (\lambda'(m') \in \mathcal{L} \rightarrow \lambda(h(m')) = \lambda'(m'))$.

Observe that a containment mapping is in fact a homomorphism with respect to the parent-child and label predicates if the tree query does not contain wildcards.

We can now define the semantics of a tree query.

Definition 4. Let $P = (T, s, d)$ be a tree query, and let D be a document. The semantics of P given D , denoted $P(D)$, is defined as the set

$$\{(h(s), h(d)) \mid h \text{ is a containment mapping of } P \text{ in } D\}.$$

The semantics of \emptyset on D , i.e., $\emptyset(D)$, is the empty set.

Example 3. Figure 2 shows an example of a tree query. The semantics of this tree query given the document in Figure 1 is the set of pairs of that document exhibited in Example 2. We will show later in the paper that this tree query is actually equivalent with the Path^+ expression given in Example 2.

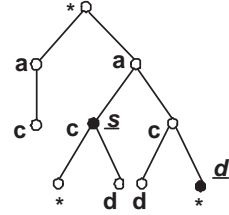


Fig. 2. An Example Tree Query.

3 Equivalences of Query Languages

In this section, we show that Path^+ , \mathbf{T} , and $\text{Path}^+(\Pi_1, \Pi_2)$ are equivalent in expressive power by exhibiting a translation algorithm that translates an expression in one language to an equivalent expression in one of the other languages.

Proposition 1. The query language \mathbf{T} is at least as expressive as Path^+ , and there exists an algorithm translating an arbitrary Path^+ expression into an equivalent expression of \mathbf{T} (i.e., a tree query or \emptyset .)

Proof. It is straightforward to translate the primitives to expressions of \mathbf{T} .

Now, let E be a Path^+ expression for which P is the equivalent expression in \mathbf{T} . If P equals \emptyset , then both $\Pi_1(E)$ and $\Pi_2(E)$ are translated into \emptyset . Otherwise, let $P = (T, s, d)$ be the tree query under consideration. Then $\Pi_1(E)$ is translated into $P_1 = (T, s, s)$, and $\Pi_2(E)$ is translated into $P_2 = (T, d, d)$.

Finally, let E_1 and E_2 be expressions for which P_1 and P_2 are the equivalent expressions in \mathbf{T} . If one of P_1 or P_2 equals \emptyset , then both $E_1; E_2$ and $E_1 \cap E_2$ are translated into \emptyset . Otherwise, let $P_1 = (T_1, s_1, d_1)$ and $P_2 = (T_2, s_2, d_2)$ the two tree queries under consideration.

(1) *Translation of composition.* First, apply the algorithm *Merge1* (Figure 3) to the labeled trees T_1 and T_2 and the nodes d_1 and s_2 . If the result is \emptyset , so does the translation of $E_1; E_2$. Otherwise, let T be the returned labeled tree. Then $E_1; E_2$ is translated into the tree query $P = (T, s_1, d_2)$.

(2) *Translation of intersection.* First, apply the algorithm *Merge1* to the labeled trees T_1 and T_2 and the nodes s_1 and s_2 . If the result is \emptyset , so does the translation

Algorithm Merge1

Input: two disjoint labeled trees
 $T_1 = (V_1, Ed_1, \lambda_1)$
and $T_2 = (V_2, Ed_2, \lambda_2)$;
nodes $m_1 \in V_1$ and $m_2 \in V_2$.

Output: a labeled tree or \emptyset .

Method:
 $q = \min(\text{depth}(m_1, T_1), \text{depth}(m_2, T_2))$
for $k = 0, \dots, q$
 if the level- k ancestors of m_1 and m_2
 have incompatible labels, **return** \emptyset
for $k = 0, \dots, q$
 merge the level- k ancestors m_1^k of m_1
 and m_2^k of m_2 into a node labeled
 $\lambda_1(m_1^k) + \lambda_2(m_2^k)$
return the resulting labeled tree.

Fig. 3. The Algorithm *Merge1*.

Algorithm Merge2

Input: a labeled tree $T = (V, Ed, \lambda)$ and
nodes $m_1, m_2 \in V$;

Output: a labeled tree or \emptyset .

Method:
let $q_1 = \text{depth}(m_1, T)$;
let $q_2 = \text{depth}(m_2, T)$
if $q_1 \neq q_2$ **return** \emptyset
for $k = 0, \dots, q_1 = q_2$
 if the level- k ancestors of m_1 and m_2
 have incompatible labels, **return** \emptyset
for $k = 0, \dots, q_1 = q_2$
 merge the level- k ancestors m_1^k of m_1
 and m_2^k of m_2 into a node labeled
 $\lambda_1(m_1^k) + \lambda_2(m_2^k)$
return the resulting labeled tree

Fig. 4. The Algorithm *Merge2*.

of $E_1 \cap E_2$. Otherwise let T_{int} be the labeled tree returned by *Merge1*. Next, apply the algorithm *Merge2* (Figure 4) to the labeled tree T_{int} and the nodes d_1 and d_2 . If the result is \emptyset , so does the translation of $E_1 \cap E_2$. Otherwise, let T be the labeled tree returned by *Merge2*. Then $E_1 \cap E_2$ is translated into the tree query $P = (T, s, d)$, where s is the node that resulted from merging s_1 and s_2 , and d is the node that resulted from merging d_1 and d_2 .

Example 4. Consider again the Path⁺ expression given in Example 2:

$$\Pi_1(\downarrow); \Pi_2(\hat{d}; \uparrow; \hat{c}); \Pi_2(\hat{a}; \downarrow; \hat{c}); \uparrow; \Pi_2(\Pi_1((\downarrow; \hat{a}; \downarrow) \cap (\downarrow; \downarrow; \hat{c})); \downarrow); \downarrow; \Pi_1(\hat{c}; \downarrow; \hat{d}); \hat{c}; \downarrow$$

We will now translate this expression into a tree query. First write the expression as $E_1; E_2; E_3$, where

$$E_1 = \Pi_1(\downarrow); \Pi_2(\hat{d}; \uparrow; \hat{c}); \Pi_2(\hat{a}; \downarrow; \hat{c}); \uparrow$$

$$E_2 = \Pi_2(\Pi_1((\downarrow; \hat{a}; \downarrow) \cap (\downarrow; \downarrow; \hat{c})); \downarrow)$$

$$E_3 = \downarrow; \Pi_1(\hat{c}; \downarrow; \hat{d}); \hat{c}; \downarrow$$

The translation is illustrated in Figure 5. Figure B exhibits how expression E_2 is translated into a tree query. Figures B.1, B.2, B.3, B.4 and B.5 correspond to the translations of the subexpressions $\downarrow; \hat{a}; \downarrow$, $\downarrow; \downarrow; \hat{c}$, $(\downarrow; \hat{a}; \downarrow) \cap (\downarrow; \downarrow; \hat{c})$, $\Pi_1((\downarrow; \hat{a}; \downarrow) \cap (\downarrow; \downarrow; \hat{c}))$, and $\Pi_2(\Pi_1((\downarrow; \hat{a}; \downarrow) \cap (\downarrow; \downarrow; \hat{c})); \downarrow)$ respectively. Figure B.6, finally, corresponds to the translation of E_2 . Figures A and C exhibit the translation of the expressions E_1 and E_3 respectively (details omitted).

Finally, Figure D exhibits the translation of the expression $E_1; E_2; E_3$.

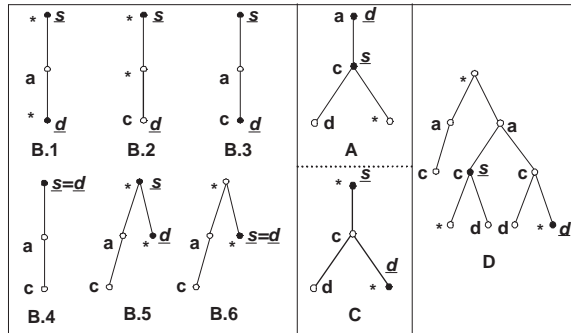


Fig. 5. Translation of the Tree Query in Exp. 4

```

Algorithm Tree_to_Path
Input: a tree query  $P = (T, s, d)$ ;
Output: a  $\text{Path}^+(\Pi_1, \Pi_2)$  expression  $E$ .
Method:
  if  $T$  is base case
     $E := \text{basecase}(T, s, d)$ 
  else if  $s$  is not an ancestor of  $d$  (case 1)
     $p :=$  the parent of  $s$ 
     $T_1 :=$  the subtree of  $T$  rooted at  $s$ 
     $T_2 :=$  the subtree resultant from removing all nodes in  $T_1$  from  $T$ 
    if  $s$  has no child and  $\lambda(s)$  is wildcard
       $E := \uparrow; \text{Tree\_to\_Path}(T_2, p, d)$ 
    elseif  $d$  is the parent of  $s$ ,  $d$  has no ancestor, no child other than  $s$  and  $\lambda(d)$  is wildcard
       $E := \text{Tree\_to\_Path}(T_1, s, s); \uparrow$ 
    else  $E := \text{Tree\_to\_Path}(T_1, s, s); \uparrow; \text{Tree\_to\_Path}(T_2, p, d)$ 
  else if  $s$  is not the root (case 2)
     $r :=$  the root of  $T$ 
     $T_1 :=$  the subtree of  $T$  rooted at  $s$ 
     $T_2 :=$  the subtree resultant from removing all strict descendants of  $s$  from  $T$ , with
       $\lambda(d)$  assigned to the wildcard *
    if  $s$  has no child and  $\lambda(s)$  is wildcard
       $E := \Pi_2(\text{Tree\_to\_Path}(T_2, r, s))$ 
    else  $E := \Pi_2(\text{Tree\_to\_Path}(T_2, r, s)); \text{Tree\_to\_Path}(T_1, s, d)$ 
  else if  $s$  is a strict ancestor of  $d$  (case 3)
     $p :=$  the parent of  $d$ 
     $T_1 :=$  the subtree of  $T$  rooted at  $d$ 
     $T_2 :=$  the subtree resultant from removing all nodes in  $T_1$  from  $T$ 
    if  $d$  has no child and  $\lambda(d)$  is wildcard
       $E := \text{Tree\_to\_Path}(T_2, s, p); \downarrow$ 
    elseif  $s$  is the parent of  $d$ ,  $s$  has no child other than  $d$  and  $\lambda(d)$  is wildcard
       $E := \downarrow; \text{Tree\_to\_Path}(T_1, d, d)$ 
    else  $E := \text{Tree\_to\_Path}(T_2, s, p); \downarrow; \text{Tree\_to\_Path}(T_1, d, d)$ 
  else if  $s = d$  is the root (case 4)
     $c_1, \dots, c_n :=$  all children of  $s$ 
    for  $i := 1$  to  $n$  do
       $T_i :=$  the subtree of  $T$  containing  $s, c_i$ , and all descendants of  $c_i$ , with  $\lambda(s)$  assigned
        to the wildcard *
    if  $\lambda(s)$  is wildcard *
       $E := \Pi_1(\text{Tree\_to\_Path}(T_1, s, c_1)); \dots; \Pi_1(\text{Tree\_to\_Path}(T_n, s, c_n))$ 
    else  $E := \Pi_1(\text{Tree\_to\_Path}(T_1, s, c_1)); \dots; \Pi_1(\text{Tree\_to\_Path}(T_n, s, c_n)); \lambda(s)$ 
  return  $E$ 

```

Fig. 6. The Algorithm *Tree_to_Path*.

Proposition 2. *The query language $\text{Path}^+(\Pi_1, \Pi_2)$ is at least as expressive as \mathbf{T} , and there exists an algorithm translating an arbitrary \mathbf{T} expression into an equivalent expression of $\text{Path}^+(\Pi_1, \Pi_2)$.*

Proof. Clearly, \emptyset is translated into \emptyset . We also have that (1) the tree query $((\{s\}, \emptyset), s, s)$ is translated to ϵ if $\lambda(s) = *$ and is translated to $\lambda(\hat{s})$ otherwise; (2) the tree query $((\{s, d\}, \{(s, d)\}), s, d)$, where $\lambda(s) = \lambda(d) = *$, is translated to \downarrow ; and (3) the tree query $((\{s, d\}, \{(d, s)\}), s, d)$, where $\lambda(s) = \lambda(d) = *$, is \uparrow . We collectively call (1), (2), and (3) the base cases, and in the algorithm exhibited in Figure 6 they are handled by the function $\text{basecase}(T, s, d)$. For an arbitrary tree query $P = (T, s, d)$, a recursive translation algorithm is exhibited in Figure 6.

Example 5. Consider the tree query in Figure 5.D. Following the algorithms in Figure 6, this tree query can be translated into an equivalent $\text{Path}^+(\Pi_1, \Pi_2)$ expression: $\Pi_1(\downarrow); \Pi_1(\downarrow; \hat{d}); \hat{c}; \uparrow; \Pi_2(\Pi_1(\downarrow; \Pi_1(\downarrow; \hat{c}); \hat{a}); \downarrow); \hat{a}; \downarrow; \Pi_1(\downarrow; \hat{d}); \hat{c}; \downarrow$.

We can now summarize Propositions 1 and 2.

Theorem 1. *The query languages Path^+ , \mathbf{T} and $\text{Path}^+(\Pi_1, \Pi_2)$ are all equivalent in expressive power, and there exist translation algorithms between any two of them.*

4 Normal Form for Expressions in the Path^+ Algebra

Normalization is frequently a critical step in rule-based query optimization. It serves the purpose of unifying queries with the same semantics, detect containment among sub-queries, and establish the foundation for cost-based query optimization, in which evaluation plans are to be generated. As it will turn out, using this normal form, we can decompose a Path^+ query into sub-queries that can be expressed in $\text{DPath}^+(\Pi_1)$, a very small sub-fragment of Path^+ for which efficient evaluation strategies are available [7]. The full query can then be evaluated by joining the results of these $\text{DPath}^+(\Pi_1)$ expressions.

When we revisit Section 3, where the translation from queries in \mathbf{T} to expressions in $\text{XPath}^+(\Pi_1, \Pi_2)$ is described, we observe that the result of the translation is either \emptyset , or ϵ , or has the following general form (composition symbols have been omitted for clarity):

$$C_{u_m} \uparrow \cdots \uparrow C_{u_1} \uparrow C_{top} \downarrow C_{d_1} \downarrow \cdots \downarrow C_{d_n},$$

where (1) $m \geq 0 \wedge n \geq 0$; (2) the C_i expressions, for $i \in u_1, \dots, u_m, d_1, \dots, d_n$, are of the form $[\Pi_1(D)]^*[\hat{l}]^?$, where the D expressions are $\text{DPath}^+(\Pi_1)$ expressions in the normal form; and (3) C_{top} is of the form $[\Pi_2(D)][\Pi_1(D)]^*[\hat{l}]^?$ where D is an expression in $\text{DPath}^+(\Pi_1)$ in the normal form. Observe that in particular, there are no \cap operations present in the normal form, and that there appears at most one Π_2 operation. We say that a $\text{Path}^+(\Pi_1, \Pi_2)$ expression of this form is in *normal form*.

Example 6. Reconsider Example 2. Clearly, the expression

$$\Pi_1(\downarrow); \Pi_2(\hat{d}; \uparrow; \hat{c}); \Pi_2(\hat{a}; \downarrow; \hat{c}); \uparrow; \Pi_2(\Pi_1((\downarrow; \hat{a}; \downarrow) \cap (\downarrow; \downarrow; \hat{c})); \downarrow); \downarrow; \Pi_1(\hat{c}; \downarrow; \hat{d}); \hat{c}; \downarrow$$

is not in normal form (e.g., note that this expression contains an intersection operation and multiple occurrences of the Π_2 operation). In Example 4, we exhibited how this expression is translated in the tree query shown in Figure 2. In Example 5, we exhibited how this tree query is translated into the $\text{Path}^+(\Pi_1, \Pi_2)$ expression

$$\underbrace{\Pi_1(\downarrow); \Pi_1(\downarrow; \hat{d}); \hat{c}; \uparrow}_{C_{u_1}}; \underbrace{\Pi_2(\Pi_1(\downarrow; \Pi_1(\downarrow; \hat{c}), \hat{a}); \downarrow); \hat{a}; \downarrow}_{C_{top}}; \underbrace{\Pi_1(\downarrow; \hat{d}); \hat{c}; \downarrow}_{C_{d_1}} \underbrace{\phantom{\Pi_1(\downarrow; \hat{d}); \hat{c}; \downarrow}}_{C_{d_2}}$$

This expression is in normal form, with the key ingredients identified below the expression.

We have the following theorem.

Theorem 2. *The tree-to-path algorithm of Figure 6 translates each tree query into an equivalent $\text{Path}^+(\Pi_1, \Pi_2)$ expression which is in normal form.*

Proof. \emptyset is translated into the expression \emptyset . The tree query with a single node labeled with a wildcard is translated into the expression ϵ .

Case 1 of the translation algorithm deals with the generation of the upward fragment $C_{u_m} \uparrow \cdots \uparrow C_{u_1} \uparrow$ in the normal form expression; Case 2 deals with generation of the optional $\Pi_2()$ expression C_{top} ; Case 3 deals with the generation of the downward fragment $\downarrow C_{d_1} \downarrow \cdots \downarrow C_{d_n}$ in the normal form expression; and Case 4 deals with the generation of the expression $[\Pi_1(D)]^*[\hat{l}]^?$ that is associated with a node in the tree query.

5 Resolution Expressiveness

So far, we have viewed Path^+ as a query language in which an expression associates to every document a binary relation on its nodes representing all paths in the document defined by that expression. We have referred to this view as the *query-expressiveness* of Path^+ . Alternatively, it can be viewed as language in which, given a document and a pair of its nodes, one wants to navigate from one node to the other. From this perspective, it is more meaningful to characterize the language's ability to distinguish a pair of nodes or a pair of paths in the document, which we will refer to as the *resolution-expressiveness* of Path^+ .

In this section, we first establish that two nodes in a document cannot be resolved by a Path^+ expression if and only if the paths from the root of that document to these nodes have equal length, and corresponding nodes on these paths are bisimilar, a property that has been called *1-equivalence* in [10]. The proof has the same structure as the proofs of similar properties for other fragments of Path in [10]. Next, we extend this result to the resolving power of Path^+ to pair of paths in a document.

We first make precise what we mean by indistinguishability of nodes.

Definition 5. Let m_1 and m_2 be nodes of a document D .

- m_1 and m_2 are expression-related (denoted $m_1 \geq_{exp} m_2$) if, for each Path^+ expression E , $E(D)(m_1) \neq \emptyset$ implies $E(D)(m_2) \neq \emptyset$, where $E(D)(m_1)$ and $E(D)(m_2)$ refer to the sets $\{n \mid (m_1, n) \in E(D)\}$ and $\{n \mid (m_2, n) \in E(D)\}$, respectively.
- m_1 and m_2 are expression-equivalent (denoted $m_1 \equiv_{exp} m_2$) if $m_1 \geq_{exp} m_2$ and $m_2 \geq_{exp} m_1$.

As already announced, we intend to show that the semantic notion of expression-equivalence coincides with the syntactic notion of 1-equivalence. Before we can give the formal definition of 1-equivalence of nodes, we need a few intermediate definitions. First, we define downward 1-relatedness of nodes recursively on the height of these nodes:

Definition 6. Let m_1 and m_2 be nodes of a document D . Then m_1 and m_2 are downward 1-related (denoted $m_1 \geq_1^{\downarrow} m_2$) if and only if

1. $\lambda(m_1) = \lambda(m_2)$; and
2. for each child n_1 of m_1 , there exists a child n_2 of m_2 such that $n_1 \geq_1^{\downarrow} n_2$.

We now bootstrap Definition 6 to 1-relatedness of nodes, which is defined recursively on the depth of these nodes:

Definition 7. Let m_1 and m_2 be nodes of a document D . Then m_1 and m_2 are 1-related (denoted $m_1 \geq^1 m_2$) if

1. $m_1 \geq^1 m_2$; and
2. if m_1 is not the root, and p_1 is the parent of m_1 , then m_2 is not the root and $p_1 \geq^1 p_2$, with p_2 the parent of m_2 .

Finally, we are ready to define 1-equivalence of nodes:

Definition 8. Let m_1 and m_2 be nodes of a document D . Then m_1 and m_2 are 1-equivalent (denoted $m_1 \equiv^1 m_2$) if and only if $m_1 \geq^1 m_2$ and $m_2 \geq^1 m_1$.

We can now establish that two nodes in a document cannot be resolved by a Path^+ expression if and only if the paths from the root of that document to these nodes have equal length, and corresponding nodes on these paths are bisimilar, a property that has been called *1-equivalence* in [10].

Theorem 3. Let m_1 and m_2 be nodes of a document D . Then, $m_1 \equiv_{exp} m_2$ if and only if $m_1 \equiv^1 m_2$.

Obviously, two nodes are bisimilar (called *downward 1-equivalent* in [10]) if they are downward 1-related in both directions. For the purpose of abbreviation, we extend 1-relatedness to pairs of nodes, as follows. Let m_1, m_2, n_1 , and n_2 be nodes of a document D . We say that $(m_1, n_1) \geq^1 (m_2, n_2)$ whenever $m_1 \geq^1 m_2$, $n_1 \geq^1 n_2$, and $\text{sig}(m_1, n_1) \geq \text{sig}(m_2, n_2)$. Next, we extend this result to the resolving power of Path^+ to pair of paths in a document. The following theorem now states the the main result about the resolution expressiveness of Path^+ .

Theorem 4. Let m_1, m_2, n_1, n_2 be nodes of a document D . Then, the property that, for each Path^+ expression E , $(m_1, n_1) \in E(D)$ implies $(m_2, n_2) \in E(D)$ is equivalent to the property $(m_1, n_1) \geq^1 (m_2, n_2)$.

The theorem states that when we find a pair (m_1, n_1) in the result of a query in Path^+ , then we are guaranteed that any pair (m_2, n_2) such that $(m_1, n_1) \geq^1 (m_2, n_2)$, will also be in the result of the query, and vice versa.

6 Efficient Query Evaluation

Minimizing algebraic expressions and rewriting queries into sub-queries for which efficient evaluation plans are available is a practice used extensively in relational database systems. An example of this is selection push down for the selection conditions on which indices are available. It is natural that the same principle and procedure is followed in XML query processing and optimization.

6.1 Minimization of Tree Queries

The results on containment and minimization of tree queries can easily be derived using the theory developed in Section 5. In particular, each tree query can be translated to a unique, equivalent, and minimal tree query.

First, we extend the notion of containment mapping (Section 2.2). Thereto, let $P_1 = ((V_1, Ed_1, \lambda_1), s_1, d_1)$ and $P_2 = ((V_2, Ed_2, \lambda_2), s_2, d_2)$ be tree queries. A *query containment mapping* of P_1 in P_2 is a mapping $h : V_1 \rightarrow V_2$ such that

1. for all $m_1, n_1 \in V_1$, $(m_1, n_1) \in Ed_1$ implies that $(h(m_1), h(n_1)) \in Ed_2$;

2. for all $m_1 \in V_1$, $\lambda_1(m_1) \in \mathcal{L}$ implies that $\lambda_2(h(m_1)) = \lambda_1(m_1)$; and
3. $h(s_1) = s_2$ and $h(d_1) = d_2$.

Proposition 3. *Let P_1 and P_2 be tree queries. Then P_2 is contained in P_1 if and only if there is a query containment mapping of P_1 in P_2 . [9]*

Now, let $P = ((V, Ed, \lambda), s, d)$ be a tree query. We define the *first reduction* of P , denoted $\text{red}_1(P)$, to be the tree query obtained from P by merging 1-equivalent nodes. For this purpose, we interpret P as a document $D = (V, Ed, \lambda')$, wherein the nodes are relabeled as follows: (1) $\lambda'(s) = \lambda(s)_s$; (2) $\lambda'(d) = \lambda(d)_d$; and (3) for all other nodes m in V , $\lambda'(m) = \lambda(m)$.

Lemma 1. *A tree query and its first reduction are equivalent.*

On the extended labels introduced above, we define an order which is the reflexive-transitive closure of the following: (1) for all $\ell \in \mathcal{L} \cup \{*\}$, $\ell_s \geq \ell$; (2) for all $\ell \in \mathcal{L} \cup \{*\}$, $\ell_d \geq \ell$; and (3) for all $\ell \in \mathcal{L}$, $\ell \geq *$.

We say that two extended labels ℓ_1 and ℓ_2 are compatible if either $\ell_1 \geq \ell_2$ or $\ell_2 \geq \ell_1$. For compatible extended labels ℓ_1 and ℓ_2 , we define $\ell_1 + \ell_2 = \max(\ell_1, \ell_2)$.

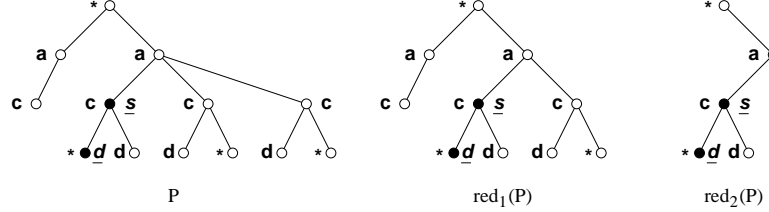


Fig. 7. A tree query and its first and second reductions.

Finally, we extend the notion of 1-relatedness from nodes of a document to nodes of a tree query with extended labels by replacing the condition $\lambda(m_1) = \lambda(m_2)$ in Definition 6 by $\lambda(m_1) + \lambda(m_2) = \lambda(m_2)$. We shall then say that m_1 and m_2 are 1-*related and denote this by $m_1 \geq_*^1 m_2$.

For P a tree query, we define the *second reduction* of P , denoted $\text{red}_2(P)$, to be the tree query by deleting from $\text{red}_1(P)$ in a top-down fashion every node m_1 for which there exists *another* node m_2 such that $m_1 \geq_*^1 m_2$. Notice that the purpose of doing the reduction in two steps is to ensure that the graph of the relation “ \geq_*^1 ” is acyclic.

Lemma 2. *A tree query and its second reduction are equivalent.*

We can now show the following.

Theorem 5. *Let P be a tree query. Every (with respect to number of nodes) minimal tree query equivalent to P is isomorphic to $\text{red}_2(P)$.*

Example 7. Figure 7 exhibits a tree query P and its first and second reductions, $\text{red}_1(P)$ and $\text{red}_2(P)$, respectively. The latter is the (up to isomorphism) unique minimal tree query equivalent to P .

6.2 Query Decomposition and Evaluation

In [7], the authors established the equivalence between the partitions on nodes (node pairs) of an XML document induced by its own structural features and the corresponding partitions induced by the $\text{DPath}^+(II_1)$ algebra. Based on these findings, they showed that, with a $P(k)$ -index [4] of $k > 1$, an index-only plan is available for answering any query in the $\text{DPath}^+(II_1)$ algebra.

We now discuss how to take advantage of this result and the normal form we discovered for the Path⁺ algebra to come up with an efficient query evaluation plan for queries in Path⁺. Consider a Path⁺ expression Exp in its normal form, represented as a tree query, in the most generic case as in Figure 8. The normal form of Exp can be written as

$$E(T_{t,s})^{-1}; E(T_{t,t}); \Pi_2(E(T_{r,t})); E(T_{t,d}),$$

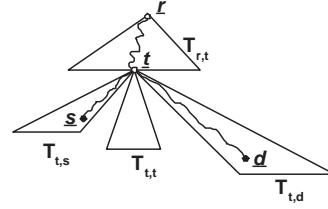


Fig. 8. General Structure of a Tree Query T

where $E(T_{t,s})$, $E(T_{t,t})$, $E(T_{r,t})$, and $E(T_{t,d})$ are the DPath⁺(Π_1) expressions corresponding to the tree queries $T_{t,s}$, $T_{r,t}$, $T_{t,t}$, and $T_{t,d}$, respectively. Since each of the four sub-queries is in DPath⁺(Π_1), efficient query evaluation with an index-only plan is available [7].

In conclusion, every Path⁺ query can be evaluated efficiently with an index-only plan provided a $P(k)$ -index [4] with $k > 1$ is available, and this with no more than three natural join operations, as guaranteed by the normal form. Indeed, for every document D , we have that

$$Exp(D) = E(T_{t,s})(D)^{-1} \bowtie E(T_{t,t})(D) \bowtie \Pi_2(E(T_{r,t}))(D) \bowtie E(T_{t,d})(D).$$

7 Discussion

This paper has been concerned with the translation of Path⁺ expressions into equivalent Path⁺(Π_1, Π_2) expressions via a tree query minimization algorithm, followed by a translation algorithm from these queries into expressions in normal form. Furthermore, it was argued that such normal form expressions have sub-expressions that can be evaluated efficiently with proper index structures.

We now generalize the Path⁺ algebra by adding set union and difference operations. Given a document specification $D = (V, Ed, \lambda)$, the *semantics* of a Path expression is defined by extending the definition of the semantics of a Path⁺ expression with

$$E_1 \cup E_2(D) = E_1(D) \cup E_2(D) \quad \text{and} \quad E_1 - E_2(D) = E_1(D) - E_2(D).$$

Both in the operations present in the Path⁺ algebra as in the set union and set difference, E_1 , and E_2 now represent arbitrary Path expressions.

The set union operation alone does not alter the resolution expressiveness results presented in this paper, since set union operations can be *pushed out* through algebraic transformation, resulting into the union of expressions which no longer contains the set union operations. The set difference operation, however, significantly increases the resolution expressiveness of the language.

Consequently, Path expressions containing set union and set difference can in general no longer be expressed as tree queries, whence our minimization and normalization algorithms are no longer applicable. Expressions containing set union, but not set difference, however, can still be normalized using the algorithms discussed in this paper, after the set union operations are moved to the top. The resulting expression will then be a union of Path⁺(Π_1, Π_2) expressions.

Ancestor/descendant relationships can be expressed in most semi-structured query languages, and have been included as \downarrow^* and \uparrow^* , in the XPath languages in some studies. However, we regard \downarrow^* and \uparrow^* merely as the transitive closure operation of the primitive operations \downarrow and \uparrow , whose characteristics have been studied in the relational context. Furthermore, with proper encoding of the data—which represent the structural relationship of a semi-structured document—the ancestor/descendant relationship can be resolved via structural join [1], which is a value join on the structural encoding.

In conclusion, the results developed for Path⁺ can be used to process more general path queries. In this regard, one can view the Path⁺ algebra to the Path algebra as one can view the project-select-join algebra to the full relational algebra.

References

1. S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE* 2002.
2. S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Tree pattern query minimization. *VLDB J.*, 11(4):315–331, 2002.
3. M. Benedikt, W. Fan, and G. M. Kuper. Structural properties of XPath fragments. *Theor. Comput. Sci.*, 336(1):3–31, 2005.
4. S. Brenes, Y. Wu, D. V. Gucht, and P. S. Cruz. Trie indexes for efficient XML query evaluation. In *WebDB*, 2008.
5. D. Chamberlin et al. XQuery 1.0: An XML query language, W3C, 2003.
6. J. Clark and S. DeRose. XML path language (XPath) version 1.0. <http://www.w3.org/TR/XPATH>.
7. G. H. L. Fletcher, D. V. Gucht, Y. Wu, M. Gyssens, S. Brenes, and J. Paredaens. A methodology for coupling fragments of XPath with structural indexes for XML documents. In *DBPL*, 2007.
8. G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. *ACM Trans. Database Syst.*, 30(2):444–491, 2005.
9. M. Götz, C. Koch, and W. Martens. Efficient algorithms for the tree homeomorphism problem. In M. Arenas and M. I. Schwartzbach, editors, *DBPL*, 2007.
10. M. Gyssens, J. Paredaens, D. V. Gucht, and G. H. L. Fletcher. Structural characterizations of the semantics of XPath as navigation tool on a document. In *PODS*, 2006.
11. R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *ICDE*, 2002.
12. C. Koch. Processing queries on tree-structured data efficiently. In *PODS*, 2006.
13. M. Marx and M. de Rijke. Semantic characterizations of navigational XPath. *SIGMOD Record*, 34(2):41–46, 2005.
14. G. Miklau and D. Suci. Containment and equivalence for a fragment of XPath. *J. ACM*, 51(1):2–45, 2004.
15. S. Pappas, J. M. Patel, and H. V. Jagadish. SIGOPT: Using schema to optimize XML query processing. In *ICDE*, 2007.