THE NATIONAL UNIVERSITY
of SINGAPORE

School of Computing
Computing 1, 13 Computing Drive, Singapore 117417

TRA4/12

Expressing and Processing Path-centric XML Queries

Huayu Wu, Ruiming Tang, Tok Wang Ling and Stephane Bressan

April 2012

T e c h n i c a l R e p o r t

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

OOI Beng Chin
Dean of School

Expressing and Processing Path-centric XML Queries

Huayu Wu
Institute for Infocomm Research, Singapore
huwu@i2r.a-star.edu.sg

Ruiming Tang
National University of Singapore
tangruiming@comp.nus.edu.sg

Tok Wang Ling
National University of Singapore
lingtw@comp.nus.edu.sg

Stéphane Bressan
National University of Singapore
steph@nus.edu.sg

ABSTRACT

The support for navigation and browsing in XML query languages is based on XPath. XQuery, a W3C standardized XML query language, introduces FLOWR constructs on top of XPath, to express more complex query purposes. However, a family of very practical queries, which aim to return or manipulate paths as first class objects, cannot be expressed by XPath or XQuery FLOWR expressions. We call them path-centric queries. Of course, a user can program an XQuery user-defined recursive function to meet path-centric queries, but the concern is the convenience and practicality of requiring normal database users to be equipped with programming skills. In this paper, we analyze the root cause of the limited expressivity of XPath and XQuery for path-centric queries. Based on the analysis, we propose seamless extension to XQuery FLOWR to elegantly express path-centric queries. We further investigate intra-path aggregation, an analytical operation in path-centric queries, and show how they can be easily expressed and efficiently processed.

1. INTRODUCTION

1.1 Context

In order to realize the World Wide Web of data, we need data models and query languages that can support the integration of semantically and structurally heterogeneous data. These data models and languages must combine the ability to formulate structured queries together with navigation and browsing capabilities.

XML and its query languages is an early instance of such models that is already in use in many applications and domains. At the core of the navigation and browsing facilities of XML main stream languages, e.g., XQuery and XSL, are the path expressions of XPath. Although these languages have been pragmatically enhanced with constructs that make them Turing complete, the core idea is to select, combine and aggregate sub-elements using path expressions, conditions and aggregate functions. This is done in XQuery, for instance with FLOWR expressions with aggregate functions.

1.2 Motivation

Proceedings of the VLDB Endowment, Vol. X, No. Y

A family of very practical queries cannot be conveniently expressed in XPath or XQuery. These queries intend to return or manipulate paths as first class objects. They intend to use path expressions as patterns matching answers, which are themselves paths, rather than as conditions for selecting entire sub-trees. We call them *path-centric* queries.

Let us, for illustration purposes, consider an XML document containing TreeBank [24], a parsed corpus of English texts annotated with their syntactic structure using XML tags. Such an XML document is illustrated in Fig. 1(a). One user, a linguist, may legitimately be interested in finding the nesting of prepositional phrases ("PP") enclosing the word "front" in verb phrases ("VP") in order to study grammatical patterns. He/She could naturally think of expressing this query in XPath as:

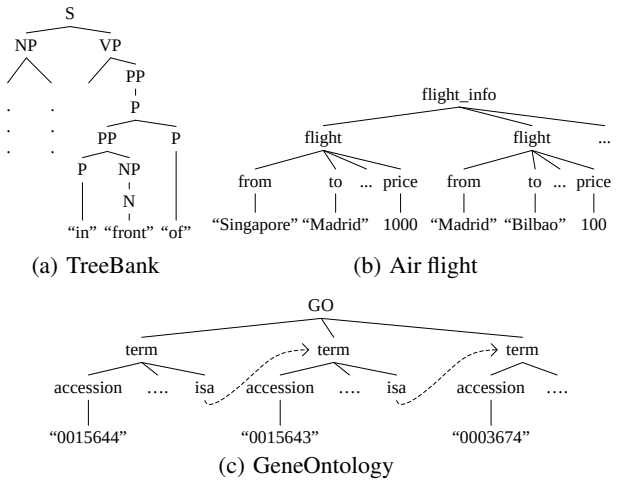
```
//VP[//PP//*/text()="front"]
```

However such a query in XPath, or corresponding queries in XQuery, would return a collection of "VP" elements, that is entire sub-trees rooted at a "VP" node. Yet the user is only interested in the paths between "VP" and "front". Without being able to project an interesting path, many path-centric query purposes cannot be easily met. For example, the user could not straightforwardly express a query counting, for every word "front" contained in a verb phrase, the number of prepositional phrases nesting it in the same verb phrase using the FLOWR constructs in XQuery. The following query cannot aggregate the number of "PP" nodes in each path between "VP" and "front". If a "VP" contains two "front", the number of "PP" in the two paths will be summed up by this query.

```
FOR $e IN doc("TreeBank.xml")//VP
LET $f := $e//PP
WHERE $f//*/text() = "front"
RETURN {$e, count($f)}
```

Actually XQuery is Turing complete for any queries only with the companion of user-defined functions to make it an XQuery program. The above query can be expressed in XQuery by introducing a recursive function, as follows:

```
declare function local:path-aggregate
  ($inits as element()*) as xs:integer
{
  for $init in $inits
    if($init/text()="front") then 0
    else for $child in $init/*
      if($child.name()="PP")
      then local:path-aggregate($child)+1
      else ()
}
local:path-aggregate
              (fn:doc("TreeBank.xml")//VP)
```



(a) TreeBank (b) Air flight

(c) GeneOntology

Figure 1: Three documents to illustrate structural path and semantic path

By doing this, we actually expect an XML database user to be equipped with programming skill (at least for recursive function), for such a simple query purpose. It will be appreciative if the basic form of XQuery, i.e., FLOWR, can be extended so that such path-centric queries can be easily expressed in a more declarative way, without using user-defined functions.

We now further analyze the root cause of the difficulty in expressing path-centric queries in XPath and XQuery. Indeed, the semantics of XPath queries, and consequently that of XQuery queries and queries of other languages leveraging XPath, is defined in terms of *elements* and *sub-elements*. XPath queries return elements, namely entire sub-trees. XPath queries do not return paths matching the path expression. Generally, XPath, XQuery and other XPath leveraged languages do not conveniently allow to prune branches of sub-trees other than by manual reconstruction (as, for instance, it is possible in the RETURN clause of an XQuery query). In other words and informally, they can perform selection but only some restricted forms of projection. This makes further queries involving the paths, such as intra-path aggregation and semantic join for path generation, difficult to express.

The following example illustrates the case of paths and semantic joins. Let us consider the XML document outlined in Figure 1(b). The XML document contains information about flights including their origin, destination, and ticket price. One user may be interested in finding the routes from Singapore to Bilbao. One way to obtain the desired result is to write an XQuery program that explicitly iterates over the different "flight" elements and tests the connections. There is no simple XQuery FLOWR query that can produce this result.

Similar queries are natural in the presence of XML ID references where IDREF attributes reference ID attributes. This is the case, for instance, in the GeneOntology data [11], as illustrated in Figure 1(c), where genes reference their super-class genes. A query to find all other family genes between two known genes cannot be expressed by the current XQuery FLOWR constructs.

1.3 Contribution

We propose seamless extensions to XQuery that can elegantly express queries that can return paths and perform intra-path aggregation. To the best of our knowledge, this is the first work studying how to project paths and how to perform intra-path aggregation in XQuery in particular and in any XML query language in general. Although some existing work (e.g., [22]) talks about projection, similar to the RETURN clause of XQuery, it can only project limited types of paths that are constructible based on explicit pattern. More discussions are presented in Section 6. The contributions can be summarized as follows.

- We address an expressivity problem, i.e., unable to express path-centric queries of XQuery FLOWR constructs. To the best of our knowledge, this is the first work pointing out this limitation.
- We extend the FLOWR constructs in XQuery with the function `SPath`, so that this basic form of XQuery can easily express path-centric queries. With our extension, one does not need to program user-defined functions to meet such a practical query purpose.
- We show that the extension of the `SPath` function is also generalized to express semantic path with different document nodes semantically linked as a chain. Then the previous queries issued to the flight document and the gene document can be easily expressed.
- We investigate a practical case, intra-path aggregation, and demonstrate how our `SPath` extension works with FLOWR constructs to express this query purpose. We also discuss and design algorithms for performing intra-path aggregation.
- Finally we conduct experiments to show the usability of our XQuery extension, and also to show the efficiency of our algorithms for intra-path aggregation.

1.4 Organization

The rest of the paper is organized as follows. We describe the background knowledge of XML data and queries in Section 2. In Section 3, we extend the XQuery with the SPath function to express stem paths (defined later) in XML data. In Section 4, we investigate a practical path-centric query purpose, i.e., intra-path aggregation, and illustrate how our XQuery expresses such a query purpose. We also discuss the execution of intra-path aggregation. Section 5 is the experimental study. The related work is revisited in Section 6. Finally we conclude this paper in Section 7.

2. BACKGROUND

2.1 XML data

An XML document is generally modeled as an ordered tree, without considering the ID references. In an XML tree, nodes represent the elements and values in the corresponding document and edges reflects the parent-child (PC) relationship between each pair of adjacent and nested elements. Any two nodes in an XML tree that are reachable to each other through a set of intermediate nodes and edges are said in an ancestor-descendant (AD) relationship. Two nodes in AD relationship, together with all intermediate nodes and edges connecting them form a path. We call it *structural path*. Besides structural path, in this paper we also investigate another type of path, *semantic path*, which is not explicitly reflected by PC edges in an XML tree, but formed by a set of document nodes logically and semantically linked as a path. More details on structural path and semantic path are presented in Section 3.

2.2 XML Query

In this paper, we use XQuery as a representative XPath-based query language and propose extension onto XQuery. As illustrated in Fig. 2, XQuery is composed of the basic form and user-defined functions. The basic form includes the XPath expressions which select document elements based on path constraints, FLOWR constructs that work on the XPath-selected elements, some extended

constructs (e.g., group-by) to supplement FLOWR, and some built-in functions such as name(), text() and aggregate functions. To simplify the description, we group the components except built-in functions in the XQuery basic form as Extended-FLOWR. As stated in Section 1, XQuery is Turing complete, thanks to the user-defined functions that give users flexibility to program his complex query purpose. However, to design a user-defined function sometimes is not easy, and many database users may not have much programming experience. In practice, the basic form with FLOWR constructs is much more welcomed by users, though its expressivity is limited (cannot express all query purposes).

XPath plays the key role in XQuery FLOWR queries. It specifies a path constraint, allowing each node along the path to have branching predicates, and navigates through the corresponding XML tree to find matched patterns. The output of an XPath query is either a set of values or a set of elements including their sub-elements, as specified by the last node in the XPath expression. Later, the FLOWR constructs work on the selected values or elements for further operations, such as filtering, iteration, join, grouping, etc. Since in XQuery, the information unit is element, which is modeled as a tree enclosing sub-elements and values, and there is no explicit operator that can cut out unwanted information in an element tree, path projection can hardly be achieved in the basic form of XQuery. The only way to do this is to use the built-in function name() to get the tag name of the elements, and to use RETURN clause to manually construct a path with tag names. This process is awkward, and furthermore, it requires the knowledge of every node along the desired path. Unfortunately, in most path projection queries, this information is exactly what the user wants to find.

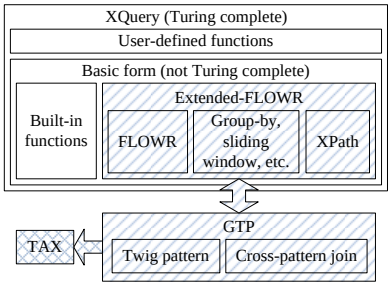


Figure 2: Relationship diagram for XML query languages

For query processing concerns, some works abstract the Extended-FLOWR expressions to tree-based expressions (as shown in Fig. 2). Generalized tree pattern (GTP) [9] is the typical work, which transforms an Extended-FLOWR query into a GTP for query plan generation. A GTP contains a set of tree patterns and join conditions between the tree patterns. A tree pattern is also called twig pattern, and it is widely considered as the core pattern for XML queries. Over decades, there are considerable research works [13] focusing on matching twig patterns to XML documents. This pattern matching is defined as selection in TAX (Tree Algebra for XML) [15], which is the fundamental algebra for GTP query processing. In fact, the research efforts in TAX selection, i.e., pattern matching already offer efficient ways for an XQuery engine to process path projection. The expressivity of XQuery becomes a bottleneck preventing this practical query purposes being effectively issued. One can imagine that an XQuery user-defined function must be programmed by the user in order to issue a path query, then the query engine needs to analyze the program and finally (if possible) knows the path pattern queried by the program, before a simple path pattern matching can be performed.

3. SPATH: EXTENDING XQUERY FOR PATH-CENTRIC QUERIES

We aim to extend the basic form of XQuery with a declarative way to project path. Only after specific paths are projected, those path-centric queries, such as intra-path aggregation, can be performed. Furthermore, we also plan to generalize the concept of "path" as a chain of document nodes that are not only structurally connected, but also semantically connected, without affecting the expressivity of path-centric operations. To do this, we must introduce a new output unit with finer granularity than element. We call it *stem node*.

3.1 Definition of stem node and stem path

An XML tree is composed of many paths. We have shown that there are many practical queries centered at each selected path. In some cases, a query may be interested in the information represented by the nodes along a path, e.g., the node name; but in some other cases, a query is interested in the information stored under each node along a path, e.g., the descendant element or values of each node. In the latter case, actually the information of a path together with the information under each node along the path should be modeled as a tree, instead of a path anymore. It is similar to XPath expression, in which the core nodes connected by "/" and "//" axes form a path, but each node may have branching predicates in "[ ]". To simplify the explanation, we introduce the concept of *stem path* to present a chain of nodes, each of which may also contain other descendant nodes.

DEFINITION 3.1. *(Stem Node) Given an XML tree T and a twig pattern t, the root of each subtree matching t in T is called a stem node.*

DEFINITION 3.2. *(Stem Path) A stem path is a list of stem nodes, in which each stem node is linked to its neighbors according to the parent-child relationship or semantic relationship.*

To output a stem node, the corresponding twig pattern whose root is modeled by the stem node will be transformed into a well-formed XML fragment and outputted. To output a stem path, all stem nodes in the stem path will be outputted based on their order in the stem path.

Stem paths is classified into structural paths and semantic path, according to how the adjacent stem nodes are linked along the stem path.

DEFINITION 3.3. *(Structural Path, Semantic Path) In a stem path, if the condition to link each pair of adjacent nodes is the structural constraint of parent-child relationship in the document tree, the stem path is called a structural path; if the condition is semantics-based join, the stem path is called a semantic path.*

In the TreeBank data in Fig. 1(a), all paths are structural paths with useful information. In the GeneOntology data and the flight data in Fig. 1(c) and 1(b), the structural paths do not contain much useful information, instead, the semantic paths generated by ID references and value-based joins are quite meaningful.

3.2 Expressing stem paths

To more declaratively and conveniently express stem path projection, we propose a function onto XQuery basic form, i.e., FLOWR.

XQuery binds variables to elements that are selected by XPath. That is the root cause of the difficulty in expressing stem paths. We propose a function `SPath` to return a set of stem paths. Then the FOR and LET clauses can be used to bind variables to stem paths, to output or to post-process.

DEFINITION 3.4. *(Definition of SPath function)*

SPath function ::= SPath$[\![tp \times tp \times tp \times cond \rightarrow sp\_set]\!]$
tp ::= XPathExpr ('/' | '//') tpRel
tpRel ::= nodeName |
*tpRel ('[' ('//')? tpRel | XPathExpr op value ']')**
cond ::= 'parent/child' | 'parent≡child' |
'parent' XPathExpr op 'child' XPathExpr
op ::= '=' | '≠' | '>' |'<' | '≥' | '≤'
sp_set ::= {*stem_path**}

where: XPathExpr *is the simplified XPath expression with only axis of '/' and '//',* nodeName *is the XML tag label,* **parent** *and* **child** *are two reserved keywords to represent the parent stem node and child stem node in a pair of adjacent stem nodes, and* value *denotes constant string or numeric values.*

DEFINITION 3.5. *(Semantics of SPath function) We use first-order logic to explain the semantics of the SPath function. Let* sp *be the stem path, in terms of a list of stem nodes as defined earlier, where* sp[first] *(*sp[last]*) denotes the first (last) stem node in the stem path* sp. sp_set *is the set of stem paths to be returned. Let the predicate* sat(X, Y) *mean* X *satisfies the constraint specified in* Y *and the rel(X, Y) mean the linking relationship between stem node* X *and* Y.

SPath(nodePattern, start, end, linkCond) ::=
{*sp | sat(sp[first], start) ∧ sat(sp[last], end) ∧*
(∀ n ∈ sp-{*sp[first], sp[last]*}*, satisfy(n, nodePattern) ∧*
satisfy(rel(n.previous, n), linkCond) ∧
satisfy(rel(n, n.next), linkCond))}

As illustrated by the semantics of the `SPath` function, stem paths are specified by four parameters. *nodePattern* describes the twig pattern rooted at each stem node along a satisfied stem path. When the stem path is a structural path, sometimes it is not necessary to specify the tag name of the stem node. In this case the input value for *nodePattern* will be a wildcard, possibly with descendant nodes that are interested by the query shown as predicates (e.g., //*[...]...). However, if semantic path is requested, it is necessary to describe what kind of stem nodes is used to construct the semantic path. *start* and *end* describe the twig patterns of the first and the last stem nodes of each satisfied stem path. All these three twig pattern parameters are expressed by XPath-like syntax. The difference is that the last test node of an XPath express specifies the return information, but in our expression, all other test nodes except the stem node are expressed within "[ ]". In other words, the XPath-like expression is only for describing the twig pattern to select the stem node, without additional semantics.

The last parameter, *linkCond* is the condition to link each pair of adjacent stem nodes in the stem path. Two pre-defined variables, ***parent*** and ***child*** stand for the parent stem node and the child stem node in a pair of adjacent stem nodes. For a structural path, this condition is simply *parent/child*, meaning that every pair of adjacent stem nodes must be in '/' (parent-child) relationship in the document tree. For a semantic path, this condition actually specifies how *parent* and *child* are joined. Thus, for a semantic path, *linkCond* will be a Boolean expression, which is similar to the join condition in the WHERE clause.

Projecting particular nodes within an XML (sub-)tree will be a special case of structural path projection. To project nodes, in the `SPath` function, *start* and *end* must be of the same node type. In addition, we preserve another operator between *parent* and *child* in *linkCond*, *parent≡child*. The "≡" sign is used to specify the equivalence of nodes. One example node projection query is to project all PP nodes in the TreeBank document. The `SPath` function will be *SPath(*,//PP,//PP,parent≡child)*. It is different from the projection of paths that start and end both at a PP-typed node, which is *SPath(*,//PP,//PP,parent/child)*, though the result of this function call also includes the single *PP* nodes.

3.3 XQuery extension

XQuery uses XPath to specify interesting elements, each of which corresponds to a subtree of the original document. The SPath function can be appended to an XPath expression, to find all stem paths under the element selected by the XPath expression. Thus in the extended XQuery, the path expressions may include the SPath function.

DEFINITION 3.6. *(Extended Path) Let PathExpr be the original XPath expression (or some built-in function returning element) used in XQuery, whose syntax and semantics is defined in XQuery specification [25] and will not be repeated here. Let ExtPathExpr be the extended path expression.*

ExtPathExpr ::= PathExpr |
PathExpr '.SPath(' tp ',' tp ',' tp ',' cond ')'

where tp *and* cond *are defined in Definition 3.4.*

One special element, the root element of an XML document is normally specified by the built-in *doc(id)* function in XQuery. To simplify the explanation, in the following examples we do not include additional selection constraints but only consider the root element and emphasize on the SPath function. SPath from a random XPath selected element can be easily expressed by replacing doc(id) function by the corresponding XPath expression. The following example illustrates how desired structural path and semantic path are found by the SPath function under an element.

EXAMPLE 3.1. ***Structural path.*** *In the TreeBank document in Fig. 1(a), all structural paths starting at "VP" and ending at some POS tag with child value of "front" are specified by*

```
doc("TreeBank.xml").SPath(*, //VP,
       //*[text()="front"], parent/child)
```

Semantic path. *In the GeneOntology document in Fig. 1(c), the semantic paths between term "0015644" and term "0003674" can be specified as*

```
doc("GeneOntology.xml").SPath(
                        //term[accession][isa],
                  //term[accession="0015644"],
                  //term[accession="0003674"],
                  parent/isa =child/accession)
```

In the Flight document in Fig. 1(b), all routes from Singapore to Bilbao (the price of each flight must be outputted for analytical purpose) can be found by

```
doc("Flight.xml").SPath(
                    //flight[from][to][price],
                  //flight[from ="Singapore"],
//flight[to="Bilbao"], parent/to =child/from)
```

Because each stem path is defined as an ordered list of stem nodes, in the extended XQuery FOR and LET can be used to iterates over a stem path and bind variables to stem nodes.

EXAMPLE 3.2. *The XQuery expression and result for the query to find the structural paths starting at "VP" and ending at some tag with child value of "front" in the document in Fig. 1(a) are:*

```
FOR $p IN doc("TreeBank.xml").SPath(*, //VP,
           //*[text()="front"], parent/child)
RETURN
    <structural_path>
      FOR $e IN $p RETURN $e
    </structural_path>
```

Result:

```
<result>
  <structural_path>
    <VP/><PP/><P/><PP/><NP/><N/>
  <structural_path>
  ...
</result>
```

If the user is only interested in outputting the "PP" nodes (e.g., for aggregation purpose) in each satisfied stem path, the RETURN clause can be:

```
RETURN
    <structural_path>
      FOR $e IN $p WHERE $e.name()="PP"
      RETURN $e
    </structural_path>
```

EXAMPLE 3.3. *The XQuery expression and result for the query to find all transit flights from Singapore to Bilbao in the flight document in Fig. 1(b) are:*

```
FOR $p IN doc("Flight.xml").SPath(
                    //flight[from][to][price],
                   //flight[from="Singapore"],
 //flight[to="Bilbao"],parent/to =child/from)
RETURN
   <semantic_path>
      FOR $e IN $p RETURN $e
   </semantic_path>
```

Result:

```
<result>
 <semantic_path>
   <flight>
     <from>Singapore</from>
     <to>Madrid</to>
     <price>1000</price>
   </flight>
   <flight>
     <from>Madrid</from>
     <to>Bilbao</to>
     <price>100</price>
   </flight>
 <semantic_path>
 ...
</result>
```

The advantage of introducing the concept of stem path into XQuery is that (1) single paths can be projected from element trees, and then (2) many intra-path query purposes can be met, as illustrated in Section 4.

3.4 Enumerating stem paths

In this section, we discuss how to extend a query processor to enumerate stem paths, i.e., executing the SPath function. Different types of XQuery processors are designed based on different XML query processing approaches, e.g., MonetDB [3] is based on relational approach, IBM DB2 [14] is based on navigational approach, etc. Different XML query processing approaches are reviewed in [13]. We focus on extending the structural join based approach, which is proven more efficient than other approaches for general cases [13][1], and has attracted most research interest.

As defined in the previous section, the SPath function should return all stem paths as specified, each of which should contain all stem nodes inside. Structural path and semantic path have different characteristics, though we discuss both of them in this paper. In a structural path, the stem nodes are connected by parent-child edges, but in a semantic path, the stem nodes are connected by value-based joins. To enumerate stem paths and their stem nodes, we have two approaches, which are suitable for structural path and semantic path respectively.

A1: from path to node. In this approach, we will find the stem paths first, based on the starting node and the ending node specified in the `SPath` function; then for each stem path, we can enumerate all its stem nodes. This approach is suitable for structural path under the structural join algorithms. Given a starting node *S* and an ending node *E*, any structural join algorithm can be adopted to easily find the structural path(s) by performing a structural join for the XPath query `//S//E`, under the help of any labeling scheme. Some labeling schemes, e.g., prefix scheme, can further help to identify all stem nodes along each structural path. However, this approach can be hardly applied to semantic path, because a semantic path is not formed by structural join for PC or AD edges.

A2: from node to path. In this approach, we will find all satisfied stem nodes first, based on *nodePattern* in the `SPath` function. Then based on *linkCond*, relevant stem nodes will be joined to construct the corresponding stem path. Obviously, this is a straightforward approach for semantic path. Theoretically, it also works for structural path, but in practice, many structural paths do not specify *nodePattern*, such as the function `SPath(*,//VP,//*[text()="front"],parent/child)` shown in Example 3.1. In this case, we have to generate all document nodes (because of * as the *nodePattern*) as potential stem nodes, and then try to connect each pair of them based on `parent/child`. Thus, this approach is not good for structural path.

According to the analysis above, the most efficient way is to adopt different approaches for different kinds of stem paths. The type of the stem paths specified by an `SPath` function can be easily told by the *linkCond* parameter, i.e., whether it is `parent/child` or some join condition. If the function returns structural paths, as mentioned above, an AD structural join is performed on the `start` and `end` nodes specified in the SPath function, to get all structural paths in terms of labels of the first and last stem nodes. Then the labels of intermediate nodes can be easily referred. If the function returns semantic paths, structural join algorithms will be applied to match the *nodePattern* to the document, to find all potential stem nodes. Then the *linkCond* will be applied to the pattern matching result to form paths. The detailed algorithms are omitted in this section, but an optimized algorithm that incorporates with aggregate functions will be presented in Section 4.2.

4. INTRA-PATH AGGREGATION: A PRACTICAL CASE

Now we discuss a practical query purpose, intra-path aggregation, as an example to illustrate the importance of the proposed

[1]Comparison was done in literature only. We did not identify any comprehensive comparison across commercial XML query engines.

SPath function in expressing path-centric queries with the basic form (FLOWR) of XQuery. We also discuss the execution of intra-path aggregation. In particular, we focus on the intra-path aggregation for structural path. We propose algorithms to perform intra-path aggregation for a large amount of structural paths returned by the SPath function, with only necessary stem nodes enumerated in each path. Later, we also leverage the existing work to perform intra-path aggregation for semantic paths.

4.1 Expression

Grouping and aggregation are very useful in analytical queries. Although the new version of XQuery has introduced the new construct to support grouping and aggregation, limited by the XPath answers, such construct only aggregates the values or elements at the end of the paths that are matched by the corresponding XPath constraints. In this point of view, we can call such an aggregate operation *inter-path* aggregation, i.e., aggregating a set of path-based selection results. However, many query purposes aim to aggregate nodes along a single path, which is referred as *intra-path* aggregation. Since the existing XQuery language has difficulty in expressing a single path, it also cannot express intra-path aggregation. We will show how we can use the extended `SPath` function to XQuery to express intra-path aggregation.

The `SPath` function returns a set of stem paths satisfying the parameter constraints. We can use the FOR clause to iterate over stem paths. Then along each stem path, we can use the FOR clause again, to iterate over the stem nodes for intra-path aggregation.

One important feature of XML is that the element tags are normally meaningful and interpretable by humans. Thus an intra-path aggregation can either work on values or work on element tags. However, since tag labels are usually not numeric, to aggregate element tags, the only function can be performed is *count( )*. For values, similar to the aggregation in SQL, any functions including *max( )*, *sum( )*, *avg( )*, etc. can be performed as long as the values are numbers. Furthermore, since in a semantic path stem nodes normally belong to the same class of interest and are linked by value-based joins, aggregation normally happens to values associated to each stem nodes. We did not identify any meaningful tag aggregation for semantic path. Thus the scope of intra-path aggregation investigated in our paper is summarized as follows.

	Tag Aggregation	Value Aggregation
Structural Path	count( )	all functions
Semantic Path	n.a.	all functions

Table 1: Classified intra-path aggregation functions

We use two examples to illustrate how to express intra-path aggregation for both element tags and values.

EXAMPLE 4.1. *(**Tag Aggregation**) Consider the TreeBank data in Fig. 1(a), and the query to find the total number of* PP *in each path that starts at* VP *and ends at some tag with value of "front". Note that it is possible that in the document there are many such paths. Thus the aggregate result should be returned for each of these paths. The XQuery expression with our extended functions is shown as:*

```
FOR $p IN doc("TreeBank.xml").SPath(*, //VP,
          //*[text()="front"], parent/child)
RETURN {
      count(FOR $e IN $p
            WHERE $e.name()="PP"
            RETURN $e)
       }
```

In the outer FOR clause, `SPath` *returns a set of stem paths and $p iterates over these stem path. In other words, in each iteration, the variable is bound to a* stem path*, rather than the element returned by the doc() function. In the nested FOR clause, $e iterate over the stem nodes of a certain stem path, and is bound to each* stem node*. Only with these new output units, intra-path aggregation for path nodes is expressible under FLOWR constructs.*

The new GROUP BY construct introduced in XQuery 3.0 can also be used in intra-path aggregation. If the query asks for the total number of different syntactic tags in each path starting at VP and ending at a tag with value of "front", the query will be:

```
FOR $p IN doc("TreeBank.xml").SPath(*, //VP,
          //*[text()="front"], parent/child)
RETURN {
 FOR $e IN $p
 GROUP BY $e.name()
 RETURN <{$e.name()}>count($e)</{$e.name()}>
       }
```

EXAMPLE 4.2. *(**Value Aggregation**) Now we consider the intra-path aggregation for values in semantic paths (an example of intra-path value aggregation in structural path is shown in the next section). Consider the query to find the quotation of all direct and transit flight from Singapore to Bilbao in the document in Fig. 1(b). The XQuery expression is:*

```
FOR $p IN doc("Flight.xml").SPath(
                   //flight[from][to][price],
                  //flight[from="Singapore"],
 //flight[to="Bilbao"],parent/to =child/from)
RETURN {
     sum(FOR $e IN $p
        RETURN $e/price)
       }
```

If the query aims to find the rout with the minimum cost, i.e., an inter-path aggregation following an intra-path aggregation, we can simply enclose the above XQuery expression with an aggregate function min( ). If we need to output the route with the minimum cost, the corresponding XQuery query is:

```
FOR $p IN doc("Flight.xml").SPath(
                   //flight[from][to][price],
                  //flight[from="Singapore"],
 //flight[to="Bilbao"],parent/to =child/from)
WHERE sum(FOR $e IN $p
      RETURN $e/price)
      =
     min(FOR $p2 IN doc("Flight.xml").SPath(
                 //flight[from][to][price],
                //flight[from="Singapore"],
                     //flight[to="Bilbao"],
                     parent/to =child/from)
        RETURN {
           sum(FOR $e2 IN $p2
               RETURN $e2/price)})
RETURN
    <route>
        FOR $e IN $p RETURN $e
    </route>
```

4.2 Execution

Performing *inter-path* aggregation, i.e., aggregating elements from different paths, in XML queries has been studied in [12, 27]. Basically, they match a query pattern to the XML document, and then

process the matchings for aggregation. For *intra-path* aggregation, similarly, we need to produce the data based on query predicates before aggregating these data. Generally, we need to enumerate all stem paths as specified by the `SPath` function, and then for each stem path, we need to enumerate all stem nodes to be aggregated.

According to the analysis in Section 3.4, when we enumerate stem paths with stem nodes, different approaches are suitable for different types of stem path. For structural paths, starting from identifying the paths and then enumerating stem nodes in each path is a good choice, while for semantic paths, we should identify potential stem nodes and then join them to form paths. Consequently, the executions of intra-path aggregation for structural path and semantic path are independent to each other. This is how the following contents are organized.

At the very beginning, we introduce the concept of *GTwig* to ease the description of algorithms in the following parts.

DEFINITION 4.1. *(GTwig) In a query with intra-path aggregation, the twig pattern formed by the* nodePattern *in the corresponding* `SPath` *function, and the aggregate attributes, group-by attributes and other predicates if they are not specified in the* nodePattern*, is called* GTwig.

The examples of *GTwig* will be presented in the next section.

4.2.1 Intra-path aggregation in structural paths

Intra-path aggregation in structural paths can be performed by document scanning. Given a path pattern, we can sequentially scan an XML document to find all the path instances, and meanwhile compute the aggregation within each path. However, this approach is only suitable for simple document and query pattern. For the document containing many recursive tags and the queries with "//"-axis and complex predicates, document scan is not efficient. This also explains why the structural join approach becomes the mainstream approach for XML query processing. As mentioned in Section 3.4, our algorithms are based on the structural join approach.

We are facing two performance challenges for intra-path aggregation in structural paths.

Challenge 1. In a large XML document, a path projection (execution of the SPath function) may result a large number of path instances in the document. How to efficiently perform intra-path aggregation for every structural path in a large set of path instances?

Challenge 2. For each structural path, the general process of intra-path aggregation is to enumerate all stem nodes along the path before aggregating them. How to enumerate only useful stem nodes, ignoring others to improve the aggregation performance?

To tackle the first challenge, we proposed an index-aided optimization for large number of path instance, as presented later. For the second challenge, we maintain an index on top of inverted lists, which can help to address only relevant nodes for a structural path.

Inverted lists are used in every structure join based XML query processing algorithms. For an XML document, each type of element corresponds to an inverted list that stores all the document nodes in this type in terms of structural labels[2] and sorted by document order. Structural joins are actually performed on inverted lists. As we can see, this structure naturally groups document nodes by types, thus it can be leveraged to tackle the second challenge. In particular, we build a novel index on top of inverted lists, named *I*^{3}*A* (**I**ndex on **I**nverted list for **I**ntra-path **A**ggregation) index. For each pair of document node type, I^{3}A index points to the inverted lists for all node types that have node instances residing in some structural path with the starting node and the ending node defined by this pair of node type. This index is not symmetric, i.e., the pairs (U, V) and (V, U) probably point to different sets of inverted lists, or one of them does not exist in I^{3}A. Fig. 3 partially shows an example I^{3}A index for the TreeBank data.

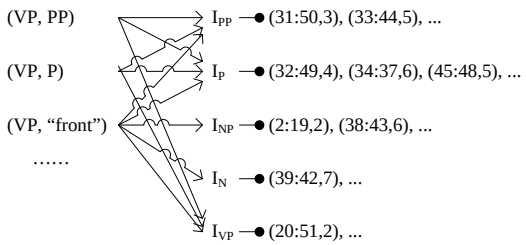


Figure 3: Example (partial) I^{3}A index for the TreeBank data

Using the I^{3}A index, given a set of structural paths with starting and ending node type, we can easily know what types of nodes are possibly contained in each path, and to address the relevant inverted lists efficiently. In the above example, given structural paths identified by a starting node VP and an ending node P, from the I^{3}A index we know that VP, PP and P nodes are possibly contained by these structural paths, and find the exact stem nodes in the relevant inverted lists. The size of the I^{3}A index is bound by $2*|T|^2+|T|*|V|$, where T is the set of different element types and V is the set of different leaf values.

As discussed, aggregation may work on either element tags or values. We describe the two cases separately.

Tag aggregation

We start from a simple case that a query aggregates nodes with a certain tag name in a structural path. Suppose after finding a set of structural paths, for a certain tag name *t*, we need to count how many *t*-nodes in each path. The basic idea is to perform an outer structural join between the set of paths (in terms of labels of the starting node and the ending node) and the inverted list of *t*, to select all *t*'s occurrences within each path. By this attempt, we can perform tag aggregation for stem nodes in each structural path without enumerating non-relevant nodes.

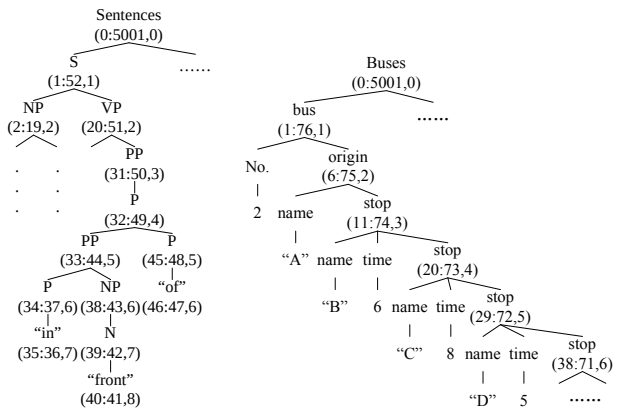
EXAMPLE 4.3. *Consider the query in Example 4.1, i.e., to find the total number of PP in each path that starts at VP and end at the value "front". The TreeBank data with labels (in containment labeling scheme [29][3]) is shown in Fig. 4(a). We first find all structural paths as specified in the function* `SPath(*,//VP,//*[text()="front"],parent/child)`. *By issuing a structural join query* `//VP//"front"`*, many pairs of labels corresponding to* VP *and* front *respectively are returned, each of which represent a structural path. (20:51,2) and (40:41,8) is one such pair. Recall that the query count the number of "PP" in each path. Then we perform an outer structural join between all tuples of labels representing paths, with the inverted list of* PP*, including (31:50,3), (33:44,5), etc. For the path starting at (20:51,2) and ending at (40:41,8), there are two and only two PP labels (31:50,3), (33:44,5) are inside the path, because of the property of containment labels, i.e., the interval (31:50) (or (33:44)) contains the interval (40:41) and is contained by the interval (20:51). Then for this path, the aggregation result is 2.*

More generally, a query may want to group all stem nodes along a structural path, and perform aggregation for each group. In this case, we can find the possible node types, i.e., possible groups,

[2]We will call it *labels* for short in the rest of the paper.

[3]Other labeling scheme, e.g., the prefix scheme, is also adoptable. In this paper, we use the containment scheme for illustration.

within every structural path by the I^3A index. Then for each node type, we perform an outer structural join between the path set (pairs of labels for starting and ending nodes) and the inverted lists corresponding to the node type. Since all inverted lists are sorted based on the labels' document order, for each structural path, all inverted lists to be joined can be scanned concurrently and many prunes can be applied. This is similar to the core idea of many structural join algorithms, e.g., TwigStack [5].



(a) TreeBank (b) BusInfo

Figure 4: XML data with containment labeling (partial)

Value aggregation

In this case, we aggregate the values under each stem node of a structural path. The basic idea is to perform a twig pattern matching for the *GTwig* before the aggregation. Note that our algorithm is orthogonal to twig pattern matching algorithms. Any state-of-the-art twig pattern matching algorithms can be adopted to match *GTwig*, and our algorithm will benefit from the efficiency of them. Again, we start from a simple case of aggregation, without grouping.

EXAMPLE 4.4. *Consider the BusInfo XML document shown in Fig. 4(b). Consider a query to find the total traveling time from stop B to stop D. The extended XQuery expression is:*

```
FOR $p IN doc("BusInfo.xml").SPath(//stop,
//stop[name="B"],//stop[name="D"],parent/child)
RETURN {
    sum(FOR $e IN $p
        WHERE $e/name/text() != "B"
        RETURN $e/time/text())
}
```

In this query, the stem node, the aggregate attribute and predicates form a *GTwig*, `//stop[name!="B"]/time` We will match this GTwig to the document to get the labels of each satisfied *stop* node and the corresponding time values. Note that we use our previous work [26] to augment twig pattern matching algorithms to extract value results. The answers of GTwig matching include the label (20:73,4) for the *stop* node and 8 for the time value, and the label (29:72,5) for the *stop* node and 5 for the time value. The desired structural path can be easily found by another pattern matching for `//stop[name="B"]//stop[name="D"]`, in the same way as that for tag aggregation. The path shown in Fig. 4(b) is one of the answers, i.e., the path from (11:74,3) to (29:72,5). Last, we perform an outer structural join between the path and the tuples of answers found by GTwig matching. In this case, both (20:73,4) and (29:72,5) are within the path from (11:74,3) to (29:72,5). Then the time value 8 and 5 are summed up as the result to this query.

Now we consider the aggregation with grouping. There are two types of groupings: one is group by the tag name of the stem nodes, and the other one is group by attribute values under the stem nodes, i.e., the values of some attributea in the GTwig. In the first case, we will scan the inverted lists once, which are achieved by the I^3A index, to classify the outer structural join results between the paths and GTwig matching answers, based on the labels of the aggregate attributes. By this way, we can group the aggregate attributes by different tag names of stem nodes, and then perform aggregation for each group. In the second case where grouping is based on the descendant attribute value of each stem node, we need to include one more post-processing step. We treat the group-by attributes the same as the aggregate attributes and get all their values during GTwig matching. Then we apply outer structural join for path and GTwig matching answers. Over the outer join result, we will perform grouping and aggregation based on the values of group-by attributes and aggregate attributes respectively, which is similar to grouping and aggregation in relational data.

Algorithm 1 Intra-path aggregation in structural paths

Input: a set of inverted lists $\{I_{e_1}, I_{e_2}, ...\}$ where e_i is an element type in the document; a collection of structural paths specified by *SPath(np, start, end, @p/@c)*; GTwig *gt* including stem node *gt.s*, an optional group-by attribute *gt.ga* and an aggregate attribute *gt.aa*; aggregate function *f*
Output: a set of aggregation result

```
1: let P be a set of structural paths in terms of the labels of the starting and ending node of each path
2: P = PatternMatching(combine(start, end))
3: initiate intermediate result sets C, J and result set R
4: if it is a tag aggregation then
5:     if f is not count() then
6:         throw exception of invalid aggregation
7:     else
8:         if gt.ga == null (no grouping) then
9:             if gt is a * node then
10:                for each path p_i in P do
11:                    compute aggregation a_i = p_i.end.level - p_i.start.level
12:                    R = R ⋃ a_i
13:            else
14:                C = I_gt
15:                J = P ⋈_s→ C
16:                R = Aggregate(J, P, null, f(gt))
17:        else
18:            let G = I^3A(start.lastTag, end.lastTag)
19:            for each tag type g in G do
20:                C = I_g
21:                J = P ⋈_s→ C
22:                R = R ⋃ Aggregate(J, P, null, f(g))
23: else
24:    //value aggregation
25:    C = PatternMatching(gt)
26:    J = P ⋈_s→ C
27:    if gt.ga == null (no grouping) then
28:        R = Aggregate(J, P, null, f(gt.aa))
29:    else
30:        if group by tag then
31:            let G = I^3A(start.lastTag, end.lastTag)
32:            for each tag type g in G do
33:                J' = J ⋈_s I_g
34:                R = R ⋃ Aggregate(J', P, null, f(gt.aa))
35:        else
36:            R = R ⋃ Aggregate(J, P, gt.ga, f(gt.aa))
37: return R
```

Procedure 2 Aggregate(Result set *J*, path set *P*, group-by attribute *ga*, aggregate function *f*)

```
1: group J by paths in P
2: for each group with path p_i do
3:     if ga == null then
4:         compute aggregation based on f
5:     else
6:         do a secondary grouping by ga, and for each subgroup compute aggregation
7: return aggregate result
```

By summing up the above discussion, the pseudo code to per-

form intra-path aggregation in structural paths is shown in Algorithm 1[4].

Index-aided optimization

In Algorithm 1, there is an outer structural join between a set of projected structural paths (in labels of the starting and ending nodes) and a set of test nodes. When the size of the XML document is large, i.e., the number of structural paths and the number of test nodes are large, this operation may be costly. Furthermore, structural join is a θ-join between labels, which is not as easy to be optimized by building internal index, as that for equi-join.

Inspired by the containment labeling scheme, the structural join is equivalent to the spatial search, based on the following proposition. Thus we propose an R-tree index-aided optimization for the outer structural join in Algorithm 1.

PROPOSITION 4.1. *Using containment labeling scheme, for a path starting at the node with the label of (a:b) (the level label is ignored) and ending at the node with the label of (c:d), a given node (x:y) is within this path iff* $x \in [a, c]$ *and* $y \in [d, b]$*, i.e.,* $a \leq x \leq c \leq d \leq y \leq b$.

For an XML document, the labels for each type of node are static. Thus for each type of element node, we build an R-tree index. Structural paths, for which intra-path aggregation is performed, will be considered as queries to search the R-tree.

EXAMPLE 4.5. *Consider the query in Example 4.3, which aggregates the number of* PP*s in each path starting at VP and ending at "front". The R-tree for the inverted list of PP is shown in Fig. 5, in which each label in the inverted list will be matched by a point in the two-dimensional R-tree space. Each path instance is encoded as a range-search rectangle in R-tree. For example, the path starting at (20:51,2) and ending at (40:41,8) corresponds to the rectangle in Fig. 5. The PP nodes within the path will be the points enclosed by the rectangle in the R-tree.*

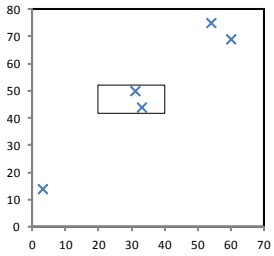


Figure 5: Example R-tree encoding with range query rectangle

In general case, the number of structural paths and the number of test nodes (e.g., the number of nodes in the inverted lists of an interesting element type) are both proportional to the document size. For a document with size of n, the cost of structural join is $O(n^2)$. However, both the R-tree construction and R-tree search time are bound by $O(n)$. Theoretically, R-tree search is more efficient than structural join for large document size. When we handle small sized XML document, performing structural join may be more efficient, as there is less overhead introduced. More comparison is illustrated by our experiments in Section 5.

4.2.2 Intra-path aggregation in semantic paths

As mentioned earlier, due to the different characteristics between structural path and semantic path, the algorithms proposed above to perform intra-path aggregation for structural paths are not adoptable for semantic paths. The major challenge here is how to form semantic paths from candidate stem nodes. As long as a set of stem nodes are clustered together to form a semantic path, the aggregation on them can be easily done.

For the query in Example 4.2, to find all transit flights from Singapore to Bilbao, we have to join many direct flights based on the equivalence between one's departure city and another's destination. If we join all the flights with themselves once, we may find the transit flights between Singapore and Bilbao via one stopover, while if we perform joins between the flights twice, we will find the transit flights via two stopovers. However, if we need to find all transit flights with unknown number of stopovers, we are not sure how many times we should join the direct flights. Moreover, there may be flights such as Singapore-Madrid-Singapore, which may cause transit flight with infinite (cyclic) stopovers.

In this section, we do not invent new algorithms, but demonstrate how the existing algorithms are leveraged to solve the problems in intra-path aggregation for semantic paths.

Approach 1: Relational Approach

Since recursive query has been well investigated in most relational database systems, the most straightforward way is to adopt such a relational approach to enumerate semantic paths and perform aggregation. For a given `SPath` function and group-by and aggregate attributes, we can have a relevant GTwig, which can be considered as a twig pattern query. The result of matching GTwig to the document is tuples of labels and child values for the relevant nodes. This result is similar to a relational table, on which recursive queries can be issued to form semantic path and to perform aggregation.

ID	From	To	Price
1	Singapore	Madrid	1000
2	Madrid	Bilbao	100
3	Singapore	Beijing	700
4	Beijing	Madrid	800
5	Singapore	Bilbao	1500
6	Madrid	Singapore	1100
…	…	…	…

Figure 6: Resulted table *Direct_flights*

For the query in Example 4.2, we first match the GTwig `//flight[from][to][price]` to the document, and get values of the three attributes for every matched pattern. Suppose the result forms a relational table *Direct_flights(from, to, price)* as shown in Fig. 6, for this query to find the total cost of each direct or transit flight, we can issue a recursive SQL query[5]:

```
WITH route(departure,arrival,stopover,
                                total_cost)AS
(
    SELECT d.from, d.to, 0, price
    FROM Direct_flights d
    WHERE d.from = 'Singapore'
  UNION ALL
    SELECT r.departure, f.to, r.stopover+1,
                        r.total_cost+f.price
    FROM route r, Direct_flights f
    WHERE r.arrival = f.from
)
```

[4]To ease the presentation, we consider a single group-by attribute and a single aggregate attribute. It is easy to extend it for multiple attributes.

[5]This query syntax is defined in IBM DB2. Most other database systems also support similar recursive queries, but may be slightly different in syntax.

```
SELECT stopover, total_cost
FROM route
WHERE arrival = 'Bilbao'
```

The relational approach is simple to implement because the core operation is actually handled by the query engine of the relational database system, which is also designed to handle relevant problems such as cycles in each path. However, the SQL query engine may not be available in an XML database system. In this case, either the whole process is implemented in the XML database system, or we adopt another approach.

Approach 2: Graph Search Approach

The graph search approach is based on some search strategies over a logical graph which is formed by the a set of candidate stem nodes and the linking condition (i.e., introducing a edge between two stem nodes, if they are satisfied by the linking condition). Basically, we will match the GTwig to the document to find candidate stem nodes, and then starting from the stem node which matches the starting node specified in the `SPath` function, we will enumerate semantic paths that end at a particular node as specified in the `SPath` function as well, by the graph search. Theoretically, by modifying some existing search strategy, e.g., breadth-first-search (BFS) and depth-first-search (DFS), we can enumerate all semantic path, and also during semantic path enumeration we can compute aggregation on-the-fly.

In practice, we notice that the number of semantic paths is quite large in many cases, and most queries do not aim to find the intra-path aggregation result for all semantic paths. Instead, most queries perform an inter-path aggregation over the intra-path aggregation result. For example, in the flight document, a travel agent does not really want to find all quotations, but wants to find the lowest quotation, the quotation of the minimum stopover, etc. To process this kind of queries, the graph search approach has advantages in early pruning.

EXAMPLE 4.6. *Consider a query to find the quotations of all the transit flights with at most two stopovers. In this case, a BFS can be adopted. From the departure city, we only need to search three levels to find the answers. In another query to find the quotations of top* k *cheapest flights, we cannot simply use BFS. In this case, a node to expand is determined by its current aggregated values on price. In other words, we will choose the node with lowest aggregated price to visit its unvisited neighbor nodes. The search will stop when we find* k *satisfied paths.*

Since graph search is a very typical problem, we do not repeat the details here.

5. EXPERIMENTS

The experiments include two parts. In the first part, we assess the usability of our SPath extension and XQuery user-defined functions to express path-centric queries. In the second part, we evaluate the efficiency of our proposed algorithms for intra-path aggregation, for structural path. Note that for semantic path, intra-path aggregation is achieved by adapting existing algorithms, thus we do not evaluate it in this paper.

5.1 Usability Evaluation

We find seven PhD students and researchers who are working on XML query to learn the syntax and usage of XQuery user-defined function (UDF) and SPath function[6]. The learning material of XQuery user-defined function is taken from W3C working draft, and the material of SPath is taken from this paper. The capacity of both materials are about the same. Then we ask them to express two path-centric queries, Q1 and Q2 for structural path and semantic path respectively, using the two extensions. We record down the time (in minute) they used to learn the two language extensions, the time to write each query, the correctness of their query expression, and their feedback on the satisfaction (difficulty) on the two extensions. Note that for the correctness, we give 2 points for answers that are correct (we tolerate minor syntax errors), 1 point for answers with minor logic errors, and 0 point for wrong answers. Table 2 shows the result. We can see that our extension takes less time in learning and query writing, and less fault-prone. It is much more satisfied by users.

Uid	Learning Time		Query Expressing				Satisfaction	
	UDF	SPath	UDF (Q1/Q2)		SPath (Q1/Q2)		UDF	SPath
			Time	Score	Time	Score		
1	10	3	20/6	1/2	5/2	2/2	2	5
2	10	5	20/15	1/1	3/3	2/2	1	5
3	30	abort	abort	abort	abort	abort	abort	abort
4	3	2	4/3	1/0	1/1	2/1	3	5
5	7	8	12/20	0/0	10/15	0/2	4	4
6	19	8	15/25	0/0	9/20	0/2	4	4
7	11	8	16/10	1/1	6/2	1/2	1	5
avg	12.9	7.2	14.5/13.2	0.7/0.8	5.7/7.2	1.2/1.8	2.5	4.7
σ	8.3	2.9	5.5/7.7	0.2/0.5	3.1/7.5	0.6/0.4	1.3	0.5

Table 2: Usability test result

5.2 Algorithm Efficiency

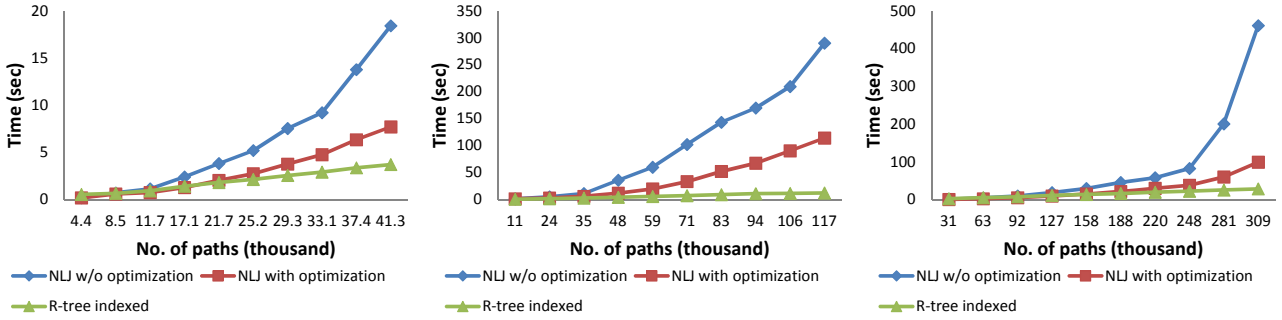
We evaluate the efficiency of our proposed algorithm for intra-path aggregation. As stated earlier, if intra-path aggregation is performed over a few paths, the efficiency will not be a problem. However, for a large XML document, a path predicate may result thousands of path instances. Thus, we test our approach under the circumstance that intra-path aggregation is performed for a large number of paths. Since we focus on intra-path aggregation, we only use simple path predicates over the TreeBank data (size varying from 5.4MB to 54.7MB) that generate large amount of path instances.

We implement three algorithms to perform intra-path aggregation for structural paths. The first algorithm is the one presented in Algorithm 1, in which the outer structural join between path instances and inverted list for the aggregate attribute is the normal nested loop join. We name it *NLJ without optimization*. In the second algorithm, we optimize the outer structural join. By noticing that the inverted list contains labels sorted by document order, for each path, we only start structurally joining it with an inverted list when a label in the inverted list falls behind the starting node of the path, and skip the rest of the labels in the inverted list once we find one label falls behind the end node of the path. We name it *NLJ with optimization*. Finally, we implement the R-tree index to avoid structural joins. All the algorithms were implemented in Java, and performed on a computer with Intel(R) Core(TM) Quad CPU with 2.83GHz, and a 3GB RAM.

The experimental results are shown in Fig. 7. In each figure, the x-axis stands for the number of path instances by executing the corresponding path query for each different-sized document, and y-axis is the total running time for intra-path aggregation.

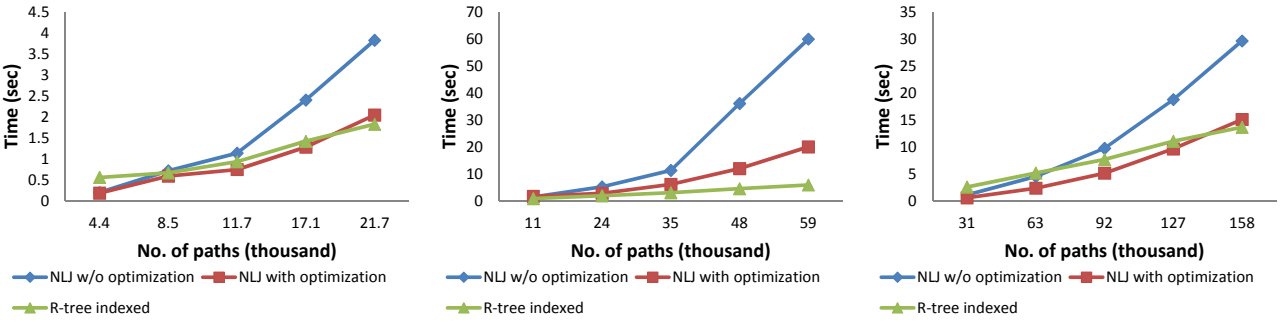
From Fig. 7, we can see that for all queries, as the document size increases, the running time of the naive nested loop join increases very fast. After optimizing the algorithm, the performance is better. Among the three algorithms, the best one is the R-tree indexed algorithm, in which the increasing rate is rather low compared to the other two. However, when we zoom in the figures, as shown in Fig. 8 (only the top 5 smallest documents are shown), we can

[6]The survey form can be found at http://www1.i2r.a-star.edu.sg/∼huwu/survey.pdf



(a) Q1: count(PP) in //NP//VP (b) Q2: count(NP) in //VP//PP (c) Q3: count(PP) in //VP//NP

Figure 7: Comparison among three algorithms for three queries



(a) Q1: count(PP) in //NP//VP (b) Q2: count(NP) in //VP//PP (c) Q3: count(PP) in //VP//NP

Figure 8: A zoomed in view for the smaller data sizes

see that the R-tree indexed algorithm is not always better than the nested loop join algorithms. The reason is that the R-tree search will introduce additional overhead. When the number of paths is not too large, R-tree search is not as efficient as the nested loop join. Since the difference between R-tree search and nested loop join is not significant for smaller number of paths, generally, using R-tree index is a better choice for intra-path aggregation, especially for a large set of paths.

6. RELATED WORK

6.1 The extensions to XML queries

There are a number of extensions to XPath and XQuery in recent years. [30] extends XPath by introducing a *Related Axis*, to specify the related relationship between query nodes. The related relationship refers to any meaningful links between document nodes, including PC relationship, AD relationship and ID reference. [6] introduces functions to express fuzzy search in XPath expression, which relaxes the strict requirement on the knowledge of XML structure for query issuers. [19] extends XPath by introducing an "until" operator, which makes XPath complete to express first-order logic expressible queries. Other XPath extensions include XPath+ [10] which extends XPath to navigate through links between XML documents (XLink), SXPath [20] allows spatial navigation for Web documents and queries.

There are also extensions to XQuery. In the initial version of XQuery, i.e., XQuery 1.0, grouping and aggregation cannot be supported. This triggers the research effort in detecting potential grouping from nested XQuery expression (e.g., [21]). However, sometimes it is very difficult or even impossible to detect such an operation. Some works try to extend XQuery by explicitly express grouping and aggregation (e.g., [4, 2]). Later, based on these research efforts, W3C drafted XQuery 1.1 which introduces GROUP BY clause in query expressions. Recently, the newest version of XQuery, XQuery 3.0, further formalize the extension in XQuery 1.1, and propose more functions, such as sliding windows, outer joins, etc. Some works extend XQuery for different applications. For example, VeXQuery [28] extends XQuery to support vector-based feature queries for multimedia XML data. They are not quite related to the theme of this paper.

To conclude, though some existing extensions enhance the expressivity of XPath or XQuery, they are still built on element selection. In other words, they improve the query power by supporting more complex element selection. A mentioned in Section 1, After finding each satisfied element, they still cannot project interesting path from the element subtree.

In [22], a new XML query language XSQuirrel is proposed, which projects a sub-document from an XML document. However, this language still cannot satisfy the query proposes mentioned in this paper. First, XSQuirrel returns a subtree that contains multiple root-to-leaf paths, which is not a general path projection (i.e., not a collection of paths starting and ending at any document nodes). Second, to project path, XSQuirrel requires the user to input the detailed pattern of the path, which is similar to the "hard coding" in the XQuery RETURN clause, as mentioned earlier. Thus, the query examples in this paper actually cannot be satisfied by XSQuirrel.

6.2 XML query processing

Since the twig pattern is the core pattern for most XML queries, there are a lot of research work focusing on efficiently matching a twig pattern query to an XML document. In the early stage, many approaches proposed to shred XML documents into relational tables and transform XML queries into SQL queries to query the database [23]. These approaches fully utilize the mature relational

query engine for XML queries, but they are inefficient for general twig pattern queries with complex structure, because to process such a query, too many expensive table joins will be involved.

Later many native approaches are proposed, among which the structural join based approach is considered the most efficient approach and attracts most research interest. In the prior works [29, 1], a twig pattern query is decomposed into binary joins, and the joins are performed sequentially. The problem is that once the join order is not well selected, there will be a large size of useless intermediate result. Bruno et al. proposed *TwigStack* [5], which is a holistic join approach to avoid producing too many useless intermediate results. It introduces a *getNext* function to ensure that each matched path is useful in later merging passes, when the path contains only AD relationships. In this point of view, *TwigStack* is optimal for twig pattern queries with only AD relationships. There are many subsequent works [16, 8, 18, 7] to optimize *TwigStack* in terms of I/O, or extend *TwigStack* to solve different kinds of problems. There are also indexing techniques to improve the efficiency of structural join [17].

7. CONCLUSION

We argue in this paper that the support for path-centric queries in XPath and XQuery is not sufficient. Indeed, in an apparent paradox, XPath queries do not return paths but rather elements, which are sub-trees. We present a family of queries that cannot smoothly be expressed in XQuery FLOWR expressions with aggregate functions and constructs. These queries require returning or manipulating paths as first class objects. We propose a simple and seamless extension to XQuery that can express such queries: the `SPath` function for path expressions. We show how XQuery with this extension can be used to effectively and elegantly express path-centric queries of interest. In particular, we use intra-path aggregation as a practical example to illustrate the importance of our extension, and also discuss query processing issues. We assess the usability of our extension and evaluate the proposed algorithms by experiments.

8. 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] K. S. Beyer, D. D. Chamberlin, L. S. Colby, F. Özcan, H. Pirahesh, and Y. Xu. Extending XQuery for analytics. In *SIGMOD Conference*, pages 503–514, 2005.

[3] P. A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *SIGMOD*, pages 479–490, 2006.

[4] V. R. Borkar and M. J. Carey. Extending XQuery for grouping, duplicate elimination, and outer joins. In *XML Conference and Expo.*, 2004.

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

[6] A. Campi, E. Damiani, S. Guinea, S. Marrara, G. Pasi, and P. Spoletini. A fuzzy extension of the XPath query language. *J. Intell. Inf. Syst.*, 33(3):285–305, 2009.

[7] S. Chen, H. Li, J. Tatemura, W. Hsiung, D. Agrawal, and K. S. Candan. Twig^{2}Stack: bottom-up processing of generalized-tree-pattern queries over XML documents. In *VLDB*, pages 283–294, 2006.

[8] T. Chen, J. Lu, and T. W. Ling. On boosting holism in XML twig pattern matching using structural indexing techniques. In *SIGMOD*, pages 455–466, 2005.

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

[10] P. C. da Silva and V. C. Times. XPath+: A tool for linked XML documents navigation. In *XSym*, pages 67–74, 2009.

[11] GeneOntology. http://www.geneontology.org/.

[12] C. Gokhale, N. Gupta, P. Kumar, L. V. S. Lakshmanan, R. T. Ng, and B. A. Prakash. Complex group-by queries for XML. In *ICDE*, pages 646–655, 2007.

[13] G. Gou and R. Chirkova. Efficiently querying large XML data repositories: A survey. *IEEE Trans. Knowl. Data Eng.*, 19(10):1381–1403, 2007.

[14] IBM DB2. http://ibm.com/db2/xml/.

[15] H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A tree algebra for XML. In *8th International Workshop on Database Programming Languages*, pages 149–164, 2002.

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

[17] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *SIGMOD*, pages 133–144, 2002.

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

[19] M. Marx. Conditional XPath. *ACM Trans. Database Syst.*, 30(4):929–959, 2005.

[20] E. Oro, M. Ruffolo, and S. Staab. Sxpath - extending xpath towards spatial querying on web documents. *PVLDB*, 4(2):129–140, 2010.

[21] S. Paparizos, S. Al-Khalifa, H. V. Jagadish, L. V. S. Lakshmanan, A. Nierman, D. Srivastava, and Y. Wu. Grouping in XML. In *EDBT Workshops*, 2002.

[22] A. Sahuguet and B. Alexe. Sub-document queries over XML with XSQirrel. In *WWW*, pages 268–277, 2005.

[23] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB*, pages 302–314, 1999.

[24] University of Pennsylvania. The penn treebank project, http://www.cis.upenn.edu/∼treebank/.

[25] W3C Consortium. XQuery 1.0: An XML query language, http://www.w3.org/TR/xquery/. 2007.

[26] H. Wu, T. W. Ling, and B. Chen. VERT: A semantic approach for content search and content extraction in XML query processing. In *ER*, pages 534–549, 2007.

[27] H. Wu, T. W. Ling, L. Xu, and Z. Bao. Performing grouping and aggregate functions in XML queries. In *WWW*, pages 1001–1010, 2009.

[28] L. Xue, C. Li, Y. Wu, and Z. Xiong. VeXQuery: An XQuery extension for MPEG-7 vector-based feature query. In *SITIS*, pages 34–43, 2006.

[29] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD*, 2001.

[30] J. Zhou, T. W. Ling, Z. Bao, and X. Meng. Related axis: The extension to XPath towards effective XML search. *J. Comput. Sci. Technol.*, 27(1):195–212, 2012.