January 2005

Extending XPath to Support Linguistic Queries

Steven Bird
University of Pennsylvania

Yi Chen
University of Pennsylvania

Susan B. Davidson
University of Pennsylvania, susan@cis.upenn.edu

Haejoong Lee
University of Pennsylvania

Yifeng Zheng
University of Pennsylvania

Proceedings of the Workshop on Programming Language Technologies for XML 2005 (PLAN-X 2005).

Extending XPath to Support Linguistic Queries

Abstract
Linguistic research and language technology development employ large repositories of ordered trees. XML, a standard ordered tree model, and XPath, its associated language, are natural choices for linguistic data storage and queries. However, several important expressive features required for linguistic queries are missing in XPath. In this paper, we motivate and illustrate these features with a variety of linguistic queries. Then we define extensions to XPath which support linguistic tree queries, and describe an efficient query engine based on a novel labeling scheme. Experiments demonstrate that our language is not only sufficiently expressive for linguistic trees but also efficient for practical usage.

Comments
Proceedings of the Workshop on Programming Language Technologies for XML 2005 (PLAN-X 2005).

Extending XPath to Support Linguistic Queries

Steven Bird*,†, Yi Chen*, Susan B. Davidson*, Haejoong Lee*, and Yifeng Zheng*
*University of Pennsylvania, †University of Melbourne
{sb,yicn,susan,haejoong,yifeng}@cis.upenn.edu

ABSTRACT

Linguistic research and language technology development employ large repositories of ordered trees. XML, a standard ordered tree model, and XPath, its associated language, are natural choices for linguistic data storage and queries. However, several important expressive features required for linguistic queries are missing in XPath. In this paper, we motivate and illustrate these features with a variety of linguistic queries. Then we define extensions to XPath which support linguistic tree queries, and describe an efficient query engine based on a novel labeling scheme. Experiments demonstrate that our language is not only sufficiently expressive for linguistic trees but also efficient for practical usage.

1. INTRODUCTION

Large repositories of text and speech data are routinely collected, curated, annotated, and analyzed as part of the task of developing and evaluating new language technologies. These technologies include information extraction, question answering, machine translation, and so forth. Linguistic databases may contain up to a billion words, along with annotations at the levels of phonetics, prosody, orthography, syntax, dialog, and gesture. For instance, Penn Treebank contains a million words of manually parsed text from the Wall Street Journal [15]. The Switchboard corpus contains $2,400$ recorded and transcribed telephone conversations, some with phonetic, prosodic, syntactic and disfluency annotations [11].

Linguistic databases consist of time-series data (e.g. texts or recordings) which represents a linguistic artifact, along with hierarchical annotations. The relationship between the primary data and its annotations is shown schematically in Figure 1. Note that the trees are ordered: siblings are sequenced by virtue of the linear ordering of the time-series data.

Despite the considerable effort expended on developing linguistic query languages (see [12] for a survey), no one has systematically investigated their efficiency. As the data grows and the analysis tasks become more complex, scalability has become a critical factor.

In general, the design of a query language must balance expressiveness and efficiency. First, it should express, as naturally as possible, the queries that the user community needs. Second, it should be optimizable, supporting query rewriting, execution planning and index selection. The goal of this work is to develop a query language for linguistic data which can express a broad range of linguistic queries and which can be implemented efficiently by exploiting the mature technology of relational databases.

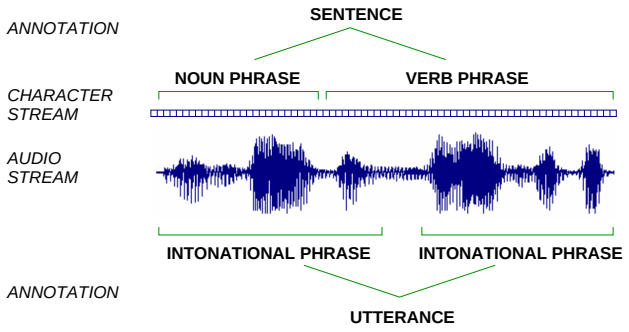
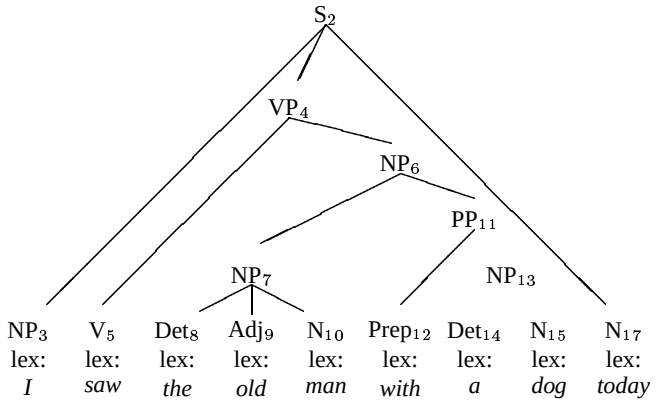


Figure 1: Linguistic Annotation: Structured Coding of Extents of Time-Series Data (e.g. Character Data, Audio Data)

Due to the reliance on an ordered tree model, a natural candidate for representing and querying linguistic data is XML together with its associated standard query languages: XPath [8] and XQuery [4]. XPath and XQuery support vertical navigations of a tree: parent, ancestor, child and descendant, and certain horizontal navigations: following and preceding. However, several horizontal navigation axes, important to linguistic queries, are lacking. It turns out that those horizontal navigations not only have practical application in linguistic queries, but also have interesting theoretical consequences for tree models. Augmenting XPath with these horizontal navigations makes the XPath axis set symmetric between vertical and horizontal navigations.

The contributions and organization of this paper are as follows. First, in section 2 we describe a new variety of semistructured data, linguistic treebanks. We analyze the data model and query requirements, and introduce a working example. Next, in section 3, we propose an expressive and intuitive linguistic query language by extending the XPath 1.0 syntax. The new language, *LPath*, supports both vertical and horizontal tree navigations in a symmetric way. In section 4 we describe a novel labeling scheme which supports efficient horizontal and vertical tree navigations. Given this language and labeling scheme, we are able to translate LPath queries into SQL queries, and leverage relational database technology for query execution, in section 5. The LPath query engine has been implemented and tested against several linguistic query engines as well as an XPath query engine. The proposed approach performs well on various data and query sets. We believe that our work has implications for XPath design and evaluation beyond linguistic



S: sentence; NP: noun phrase; VP: verb phrase; PP: prepositional phrase; Det: determiner, Adj: adjective; N: noun; Prep: preposition; V: verb; tmp: temporal (Nodes are assigned identifiers to facilitate the discussion.)

Figure 2: Tree Representation

applications. Section 6 concludes the paper with a summary and a discussion of future research.

2. DATA MODEL AND QUERY REQUIREMENTS

A common data model for linguistic annotations is an ordered labeled tree, for example, Penn Treebank [15]. Figure 2 shows the tree representation of a parsed sentence. Here the immutable terminals are the words in the sentence comprising the fringe of the tree, while the linguistic analysis is an ordered tree built over the terminals. Non-terminal nodes are annotations of sequences of terminal nodes or of other non-terminals. For instance, the node NP_7 (noun phrase) is an annotation of annotations Det_8, Adj_9 and N_{10}, denoting that 'a determiner, an adjective and a noun together compose a noun phrase.' The terminals of a linguistic tree are linearly ordered; this ordering induces an ordering on the non-terminals.

Large-scale empirical linguistics involves searching and collating tree data. We have compiled a representative sample of linguistic tree queries below, and given their results against the tree in Figure 2.

- Q_1 Find a sentence containing the word *saw*: $\{S_2\}$
- Q_2 Find noun phrases that immediately follow a verb: $\{NP_6, NP_7\}$ (both nodes immediately follow V_5)
- Q_3 Find nouns that follow a verb which is a child of a verb phrase: $\{N_{10}, N_{15}, N_{17}\}$ (all three follow V_5)
- Q_4 Within a verb phrase, find nouns that follow a verb which is a child of the given verb phrase: $\{N_{10}, N_{15}\}$ (within VP_4, N_{10} and N_{15} follow V_5)
- Q_5 Find noun phrases which are the rightmost child of a verb phrase: $\{NP_6\}$
- Q_6 Find noun phrases which are the rightmost descendant of a verb phrase: $\{NP_6, NP_{13}\}$ (both are descendants of VP_4, and no other descendants of VP_4 follow them)
- Q_7 Find all verb phrases that are comprised of a verb, a noun phrase, and a prepositional phrase: $\{VP_4\}$ (VP_4 is comprised of V_5, NP_7 and PP_{11})

S	→	NP VP (NP)	*I saw the old man* PP_{11} *today*
VP	→	V NP (NP)	*I* V_5 Det_8 Adj_9 N_{10} PP_{11} *today*
NP	→	NP PP	*I* V_5 NP_7 PP_{11} *today*
NP	→	Det Adj* N	*I* V_5 NP_6 *today*
PP	→	Prep NP	*I VP today*

(a) CFG Productions **(b) Some Proper Analyses**

Figure 3: CFG and its Proper Analyses

There are three key features of linguistic query requirements, which we will illustrate through the sample queries.

Node navigation. Linguistic trees need to be navigated in both vertical and horizontal directions. To process Q_1 which looks for a sentence with the word *saw*, we start from nodes with a tag *S*, and navigate down in their subtrees to find if there are descendants with an attribute whose name is *lex* and whose value is *saw*. This query demonstrates that linguistic trees need to be navigated vertically via hierarchical relationships, such as child, descendant, parent or ancestor relationships.

Q_2 searches for noun phrases that immediately follow a verb. Let us understand the "immediately follow" relationship in syntax tree first. We will define the relationship in the general case in the next section.

Traditionally, this relationship has been understood with respect to the context-free grammar (CFG) which licenses trees. For example, the tree in Figure 2 is a derivation tree of the context-tree grammar with production rules in Figure 3(a). We can apply grammar productions in reverse to a sentence in order to get sequences, or "proper analyses" [7]. For example, Figure 3(b) shows some proper analyses of the sentence *I saw the old man with a dog today* with respect to the grammar in Figure 3(a).

We say that a node n *immediately follows* another node m in a linguistic tree if and only if n appears immediately after m in some proper analysis according to the productions of the grammar. According to the sample proper analyses in Figure 3(b), we know that V_5 is immediately followed by NP_6, NP_7 and Det_8, and therefore we can determine that NP_6 and NP_7 are the results of Q_2.

Now consider Q_3, which involves a "follow" relationship. We say that a node n *follows* another node m if and only if n appears after m in some proper analysis. In our example, N_{10}, N_{15} and N_{17} all follow V_5.

As shown in Q_1, Q_2 and Q_3, linguistic queries involve node navigations in two dimensions: vertical and horizontal. These are commonly referred by linguists as dominance and precedence.

Subtree scoping. Node navigation sometimes needs to be scoped within a subtree. Compared with Q_3, Q_4 searches for nouns which follow a verb within a verb phrase. For example, consider a verb V_5 and the three nouns which follow it, N_{10}, N_{15}, and N_{17}. Since N_{17} is outside the verb phrase VP_4, it does not satisfy the query.

Edge alignment. Linguists are often interested in constraining the position of a constituent to be leftmost or rightmost within a particular subtree. Query Q_5 illustrates this for a child node, while Q_6 is more general. Observe that NP_6 and NP_{13} are both rightmost within VP_4.

Q_7 is a frequently-used complex query. It introduces another important notion: a sequence of nodes *comprises* some higher-level node. This notion arises from the grammar production rules and proper analyses. For example, given a proper analysis *I* V_5 NP_7 PP_{11} *today* in Figure 3(b), we can apply the production rule $NP \rightarrow NP\ PP$ in reverse to get another proper analysis *I* V_5 NP_6 *today*. Furthermore, applying the production rule $VP \rightarrow V\ NP$ in reverse on the resulting proper analysis, we get *I* VP_4 *today*. From the analyses, we can see that the sequence of nodes V_5, NP_7, PP_{11} is derived from VP_4 according to production rules. Accordingly we say that V_5, NP_7 and PP_{11} comprise VP_4. Similarly, we say that V_5 and NP_6 comprise VP_4.

After studying the requirements of linguistic queries, we find that XPath can not express Q_2, Q_4 and Q_7, since it can not express certain horizontal navigations or subtree scoping. To address these requirements, we propose a linguistic tree language, LPath, as described in the next section.

3. LPATH: A PATH LANGUAGE FOR LINGUISTIC TREES

In this section we propose a path language for linguistic trees: LPath, extending XPath. We chose XPath as the basis of our language given the focus on locating nodes in a tree instead of transforming and constructing trees. Most of the additional features of XQuery, such as node construction, iterations, joins and type checking are not required. Second, compared to XQuery, XPath has been better studied within the database community in terms of expressiveness and efficiency [10, 14]. Also various evaluation and optimization techniques for XPath have been proposed [5, 6, 13]. We start by presenting the axes of LPath which represent how to navigate from a given node to a node set in a tree. Then we define the grammar of LPath using axes as building blocks and illustrate it using the sample queries. Finally we compare LPath with the functions in XPath and XQuery.

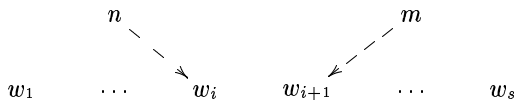
3.1 Node Navigation Axes

Since annotation trees are two-dimensional, hierarchical objects, a linguistic query requires two types of node navigation, vertical and horizontal, as explained in section 2. LPath allows vertical navigation to retrieve children and parents, and their transitive closures, descendants and ancestors, using axes. Since these axes are well-known and already defined in XPath, we do not discuss them further.

LPath supports sibling navigation to retrieve the immediately following sibling, immediately preceding sibling, and the transitive closures, following sibling and preceding sibling, using axes. Note that the immediately following sibling and immediately preceding sibling are not supported by XPath.

LPath also supports horizontal navigation to retrieve immediately following nodes. Recall from section 2 that we define the immediate following relationship between nodes using "proper analyses" of syntax trees. In fact, this relationship is generally useful for any annotation trees which do not correspond to context free grammar. Now we will generalize immediately follows as follows:

Definition 3.1: Let $w_1 \cdots w_s$ be an ordered sequence of terminals and T be an annotation tree built over the sequence. The set of nodes retrieved by applying *immediately following* axis to node n in T contains all nodes m in T, such that the leftmost terminal in m's subtree is immediately after the rightmost terminal in n's subtree. Schematically:



■

Notice that the immediately following relationship between two nodes is defined by referring to the terminal sequence. It is not hard to see that for syntax trees, the above definition is equivalent to the one defined based on "proper analyses" since the descendant terminals of a node can be derived from that node using production rules in syntax trees.

As pointed out by [14], XPath cannot express the immediately following axis. Instead, it is necessary to employ a general purpose programming language to implement this algorithm. However, both definitions given so far lead to inefficient implementations. In the next section we define a labeling scheme which permits following nodes to be found in a single step. Next we define the following axis.

Definition 3.2: Let $w_1 \cdots w_n$ be an ordered sequence of terminals and T be an annotation tree built over the sequence. The set of nodes retrieved by applying *following* axis to node n in T contains all nodes m in T, such that the leftmost terminal in m's subtree is after the rightmost terminal in n's subtree. ■

Notice that this definition of following is equivalent to the one defined in XPath: applying following axis to a node n obtains

Table 1: LPath Navigation Axes

Vertical Navigation	
/	child
//	descendant (/$^+$)
	descendant-or-self (/*)
\	parent
\\	ancestor (\$^+$)
	ancestor-or-self (\$^*$)
Horizontal Navigation	
->	immediate-following
-->	following (->$^+$)
	following-or-self (->*)
<-	immediate-preceding
<--	preceding (<-$^+$)
	preceding-or-self (<-*)
Sibling Navigation	
=>	immediate-following-sibling
==>	following-sibling (=>$^+$)
	following-sibling-or-self (=>*)
<=	immediate-preceding-sibling
<==	preceding-sibling (<=$^+$)
	preceding-sibling-or-self (<=*)
Other Axes	
.	self
@	attribute
	namespace

```
P   ::= AP | AP '{' P '}'
AP  ::=    | S AP
S   ::= A T | A T '[' R ']'
A   ::= '/' | '//' | '.' | '\' | '\\'
      | <= | => | <== | ==>
      | <- | -> | <-- | -->
T   ::= Qname | _ | '@' Qname C Qname
R   ::= R 'or' R | R 'and' R | 'not' R | '(' R ')'
      | P | P C '"' Qname '"'
C   ::= '=' | '<=' | '>=' | '<>' | 'like'
```

P: Path expression; AP: Absolute Path expression; S: Step, A: Axis; T: Tag; R: Restriction (predicate)

Figure 4: The Grammar of LPath

all nodes in the same document as n that are after n in document order, excluding n's descendants and excluding attribute nodes and namespace nodes. In contrast with the definition of following in XPath, which uses document order, we define this and all other axes on a unified data model. We define the inverse axes, immediate precedes and precedes, in the obvious way.

To retrieve nodes other than elements, we also define the axes attribute and namespace.

A summary of these LPath axes, their syntactic abbreviations, and the relationships between them, is given in Table 1. Note that the axes in vertical, horizontal and sibling direction are defined symmetrically, each of which has a primitive version and its transitive closure. We also allow 'or-self' versions of the primitive axes.[1]

3.2 LPath Grammar

We present the grammar for LPath in Figure 4. A path expression P is an absolute path optionally followed by a scoped path. The absolute path expressions AP are composed of steps S. A step consists of an axis A, a tag test T, and an optional restriction (or predicate) R. The axis A represents the navigations performed between nodes, defined in Table 1. The tag test T can be a string equality test or a wildcard '_' which matches any tag. [2] R is the restrictions introduced by '[]' to filter a node set. For each node in the set to be filtered, R is evaluated with that node as the context node. The restriction is a logical expression composed of one or more sub-expressions, connected by 'and', 'or' and 'not'.

Subtree scoping. We introduce curly braces into the language to permit subtree scopes to be expressed [3]. These will force all node navigation to be constrained to a subtree. When '{' occurs after a query node n, all the axes between '{' and '}' are evaluated within the XML subtree rooted at the XML node matching n. For example, consider Q_4 which is a scoped version of Q_3. Q_3 can be expressed as `//VP/V-->N`, and we can add the subtree scope restriction for *VP* nodes as required for Q_4 as follows: `//VP{/V-->N}`. Consider the XML tree in Figure 2: although N_{17} is a following node for V_5 in the whole tree, it is outside the scope of VP_4's subtree and so it is not part of the result for Q_4.[4]

Edge Alignment. Linguistic queries need to refer to nodes at the leftmost or rightmost edge of the subtree rooted at a specified node (e.g. Q_5 and Q_6). It turns out that we could use predicates with the following or preceding axes to specify that a given node is the rightmost or leftmost node. For example, to express Q_5, we can write an LPath query `//VP{/NP}[not<--_]`.

Since edge alignment is used extensively in linguistic queries, we introduce syntactic sugar ∧ to force left-alignment, and $ to force right-alignment. (These choices are motivated by the syntax of popular regular expression languages.) These operators are defined as follows: `^A = A[not<--_]; A$ = A[not-->_]`. Accordingly, Q_5 can be expressed as: `//VP{/NP$}`. Often ∧ and $ are used together with subtree scoping to align nodes within a subtree instead of the whole tree.[5]

3.3 LPath Examples

Now that we have discussed the syntax of the proposed language, let us consider how it can be used to represent the sample linguistic queries from section 1.

Q_1 Find a sentence containing the word *saw*.
`//S[//_[@lex=saw]`

Q_2 Find noun phrases that are immediately following a verb.
`//V->NP`

Q_3 Find nouns that follow a verb which is a child of a verb phrase.
`//VP/V-->N`

Q_4 Within a verb phrase, find nouns that follow a verb which is a child of the given verb phrase.
`//VP{/V-->N}` — compared to Q_2, Q_3 restricts the following axis navigation within the scope of the noun phrase.

Q_5 Find noun phrases which are the rightmost child of a verb phrase.
`//VP{/NP$}` — the $ operator is used to align the match to the rightmost child of a verb phrase.

Q_6 Find noun phrases which are rightmost descendants of a verb phrase.
`//VP{//NP$}`

Q_7 Find verb phrases comprised of a verb, a noun phrase, and a prepositional phrase.
`//VP[{//^V->NP->PP$}]` — notice that we require the ability to scope, express left and right alignment and immediate following. As shown in the query, ∧ forces V to align to the left edge of *VP*, and $ forces *PP* to align to the right edge.

3.4 Discussion

We have introduced LPath and compared it with XPath without functions. We can employ a user-defined function to express the immediately following navigation. An interesting question is whether we should express the extended navigation axes of LPath

[1] We are not aware of linguistic use cases for the 'or-self' variants of the axes which are part of XPath, but include them to be compatible with XPath. In the ensuing discussion, we will omit the 'or-self' versions.

[2] Instead of using $*$ to denote a wildcard to match any tag name as defined in XPath specification, we use _ as wildcard and $*$ to denote transitive closure in this paper.

[3] Note that the curly braces used here is different as the one used in XSLT for attribute value templates.

[4] Subtree scope could be expressed using variable bindings as used in XQuery. We define scope explicitly rather than use variable bindings, since it is a special case of variable binding, and enables efficient evaluation techniques comparing with general techniques required for variable bindings. We refer the readers to section 5 for more details.

[5] A more efficient evaluation for ∧ and $ is given in section 5.

as user defined functions in XPath/XQuery or define them as new axes. We want to express these node navigations as first-class citizens in a tree query language for three reasons. First, functions and axes play distinct roles in a tree language: axes define the types of node navigations in a tree, while functions usually complement the language with certain qualifiers that filter the node set. It is natural to express the node navigation relationships as axes. Second, horizontal node navigation is a common type of navigation for linguistic queries, and it is crucial that they are implemented efficiently. Finally, horizontal navigation fills a gap in the XPath axis set. The XPath axis set includes transitive horizontal navigations (follows, precedes) without defining the primitives (immediate-follows, immediate precedes). On the other hand, it includes both primitives and transitive closures for vertical navigations. It is natural to add horizontal primitives as axes to achieve 'symmetry' between vertical and horizontal navigations used in a tree.

Another question related to functions is whether or not edge alignment in LPath can be expressed using the position function in XPath. The alignment of a child node with the left or right edge of its parent can be expressed by position function in XPath. For example, Q_5 can be expressed `//VP/_[last()][self::NP]`. However, XPath cannot describe more deeply nested alignments, as required for Q_6. A putative XPath equivalent is: `//VP//_[last()][self::NP]`. However, this XPath expression evaluates to $\emptyset$ on the tree in Figure 2, while Q_6 evaluates to $\{NP_6, NP_{13}\}$. The key difference is that $\wedge$ and \$ are sensitive to node order in an XML tree, while the XPath position function considers a node's position in the sequence obtained from subquery evaluation, losing the structural information from the original XML tree.

4. LABELING SCHEME

We propose an interval-based labeling scheme to capture the structure of linguistic trees and detect relationships between tree nodes with respect to LPath axes simply by inspecting their labels.

The labeling scheme is based on the following observations for an ordered tree without unary branching (that is, each inner node has at least two children).

1. Let n be a node in a tree with a leaf sequence $w_1 \cdots w_n$, where the leaf descendants of n are $w_i \cdots w_j$, $1 \leq i \leq j \leq n$. Define $span(n) = (i, j)$.
2. Let n and m be two nodes in the tree, and suppose $span(n) = (l_n, r_n)$, and $span(m) = (l_m, r_m)$. Then n is an ancestor of m if and only if $l_n \leq l_m$ and $r_n \geq r_m$.
3. Node n immediately follows m if and only if $l_n = r_m + 1$.
4. Node n follows m if and only if $l_n > r_m$.

If we encode the order information of each node's leftmost and rightmost leaf descendants as labels, we can determine the ancestor, descendant, immediate preceding, immediate-following, preceding, and following relationships of any two nodes directly.

For a tree with unary branching, it is possible that node n and its descendant m have the same leftmost and rightmost leaf descendants, and therefore their ancestor-descendant relationship cannot be determined by this information alone. To solve this problem, we encode node depth. This depth information can also be used to distinguish the parent-child relationship from the ancestor-descendant relationship.

Table 2: Axes and Their Corresponding Label Comparisons

Vertical Navigation	
child(m, n)	$n.id = m.pid$
descendant(m, n)	$m.l \geq n.l, m.r \leq n.r, m.d > n.d$
parent(m, n)	$m.id = n.pid$
ancestor(m, n)	$m.l \leq n.l, m.r \geq n.r, m.d < n.d$
Horizontal Navigation	
immediate-following(m, n)	$m.l = n.r$
following(m, n)	$m.l \geq n.r$
immediate-preceding(m, n)	$m.r = n.l$
preceding(m, n)	$m.r \leq n.l$
Sibling Navigation	
immediate-following-sibling(m, n)	$m.l = n.r, m.pid = n.pid$
following-sibling(m, n)	$m.l \geq n.r, m.pid = n.pid$
immediate-preceding-sibling(m, n)	$m.r = n.l, m.pid = n.pid$
preceding-sibling(m, n)	$m.r \leq n.l, m.pid = n.pid$
Others	
attribute(m, n)	$m.id = n.id, m.name$begins with @

To test the sibling relationship of nodes n and m, we need to check whether they share the same parent. To expedite sibling navigations, which are frequent in linguistic queries, we assign *id* and *pid* to each node in the tree, where *id* and *pid* are the unique identifier of a node and its parent node, respectively. We also use the tag or attribute *name* of a node to distinguish element nodes from attribute nodes. We can now formalize the labeling scheme.[6]

Definition 4.1: The labeling scheme assigns each node a tuple <*left, right, depth, id, pid, name*>, shortened as <*l, r, d, id, pid, name*>, in the following fashion:

1. Let n be the leftmost leaf element. Then assign $n.l = 1$.
2. Let n be a leaf element. Then assign $n.r = n.l + 1$.
3. Let m and n be consecutive leaf elements where m is on the left. Then assign $m.r = n.l$.
4. Let n be a non-terminal element which has a sequence of leaf descendants in order: $m_1, \ldots, m_k$. Then assign $n.l = m_1.l$ and $n.r = m_k.r$.
5. For each element n, let $n.d$ be the depth of n, where the root has a depth of 1.
6. For each element n, assign a nonzero id as its unique identifier ($= f(l, r, d)$ where f is a Skolem function).
7. For each element n, assign $n.pid$ to be n's parent element identifier; if n is the root, assign $n.pid = 0$.
8. For each attribute a associated with an element n, assign the same <*l, r, d, id, pid*> as n to a.
9. For each element n, let $n.name$ be the tag name of n. For each attribute a, let $a.name$ be the attribute name of a.

■

Table 2 shows how to determine the LPath axis relationship of any two nodes by inspecting their labels.[7]

Example 4.1: Figure 5 shows the labels of the sample annotation tree in Figure 2, where the *id* attribute in the table T corresponds

[6]This definition can easily be extended to multiple trees by introducing tree identifiers.

[7]Extensions to reflexive versions of the axes are easy and are omitted here. For example, descendent-or-self(m, n) =$m.l \geq n.l, m.r \leq n.r, m.d \geq n.d$. The labels of nodes in a tree can be constructed in a single-pass using depth-first traversal. The labeling scheme is related to the data model of annotation graphs [3, 2].

left	*right*	*depth*	*id*	*pid*	*name*	*value*
1	10	1	2	1	S	
1	2	2	3	2	NP	
1	2	2	3	2	@lex	I
2	9	2	4	2	VP	
2	3	3	5	4	V	
2	3	3	5	4	@lex	saw
3	9	3	6	4	NP	
3	6	4	7	6	NP	
3	4	5	8	7	Det	
3	4	5	8	7	@lex	the
			...			
			...			

Figure 5: Relational Representation T

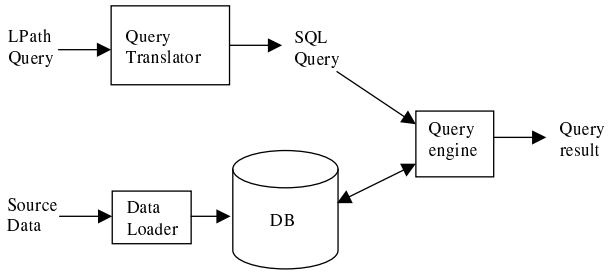


Figure 6: The System Architecture

to the node ids in Figure 2. Consider node NP_6: it has label l=3, r =9, d =3. We detect that node S_2 with label l=1, r =10, d =1 is an ancestor of NP_6 since $S_2.l \leq NP_6.l$, $S_2.r \geq NP_6.r$, and $S_2.d < NP_8.d$ according to Table 2. Furthermore, node V_5 with label l=2, r = 3, d =3 immediately precedes NP_6 since $NP_6.l = V_5.r$. ■

5. LPATH QUERY EVALUATION SYSTEM

As discussed in section 1, the two key features of a good language are expressiveness and efficiency. We have discussed the expressiveness of the proposed language with respect to the linguistic query requirements in section 3; here we will focus on efficiency.

5.1 Query System Requirements

As for the query language, there are a series of requirements to consider for the query system. These requirements motivate our decision to implement the system on top of a database engine.

Client-Server Model. Linguistic data is typically developed in the course of linguistic research and language technology development. Over time, major laboratories construct or acquire an extensive collection of this data, each stored on a central file server in its own physical format and each with accompanying tools. Research sponsors often fund the creation of linguistic data which are published and distributed widely. Researchers typically create derived versions of this data, losing provenance information. Many research methodologies would be greatly simplified if these large collections were stored on servers and accessed remotely by clients using a query mechanism which selects the required data and transforms it into the required structure. Instead of storing derived data, it would often suffice to store the query which generated it. Thus, the system should support interaction with non-local data.

Concurrent Access. Many annotation tasks require the collaboration of multiple human and automatic agents: parsers do a first-pass analysis, linguists correct the mechanically parsed data, supervisors implement quality controls, and external experts supply highly specialized annotations. Each time a new category of error is discovered, updates must be performed across the database. Further checks must be performed on newly updated data, possibly by people in different physical locations. Any system needs to support concurrent access to a shared copy of the data.

Integration. Linguistic data usually contain substantial tabular data in addition to trees, including speaker demographics and lexicons. We would like to be able to join our linguistic query expressions with queries over these auxiliary tables. For instance, for the Switchboard database, a linguist might want to restrict a query to trees over data provided by female speakers of southern US English aged 20-30 (using the demographic table), and which contain words that have sibilants (i.e. 's' and 'sh' sounds) which must found using the pronunciation table since its not obvious from spelling (e.g. *face* contains the *s* sound, and *nation* contains the *sh* sound). Furthermore, linguists often want to work within a subcollection defined by a query. All further work is then qualified by that query. Both of these needs can be met by a system which supports joins over arbitrarily complex queries.

Size. Using a high performance natural language parser it is possible to process text on the web, and permit tree queries over web data [16]. For such applications, the data will not fit in main memory. Thus the system must support efficient queries over data stored on disk.

Optimization. Given limited resources for system development, it is important to be able to exploit existing optimizations. Thus it is desirable to build the system on top of a system which does all of the standard storage and query optimizations.

5.2 System Architecture

To address these requirements, we have developed a query engine on top of a relational database engine. The LPath query evaluation system exploits the labeling scheme presented in section 4. It is composed of three modules: data loader, query translator and query engine. The data loader parses the input linguistic trees, generates a label for each node, and stores the labels into a relational database. The query translator translates an input LPath query on trees into an SQL query on tables in accordance with the labeling scheme. We use a commercial relational database as the query engine. The architecture of our LPath query processing system is presented in Figure 6.

5.3 System Description

The data loader generates a tuple $<$*left, right, depth, id, pid, name, value*$>$ for each node in a tree, and stores it into a relational table. For example, Figure 5 shows part of the relation generated for the sample annotation tree in Figure 2. Indexes are built on the table to facilitate searches.

Now we will discuss the query translator which converts an LPath query to an SQL query based on the labeling scheme. After we store the labels into a table T, each LPath axis in the query can be

evaluated as a join over T, where the join conditions involve label comparisons as shown in Table 2.

When translating LPath queries we need to handle the subtree scoping restrictions expressed using {}. Since scopes can be nested, we use a stack to keep track of them. When we encounter a node m followed by a {, we save m's label which defines the current scope on the stack, and require that any node n appearing inside the scope must be bounded by m, i.e. $m.l \leq n.l$, $n.r \leq m.r$, and $n.d > m.d$. Once the corresponding } is met, we pop the current scope from the stack.

The details of the query translation algorithm can be found at [1].

Example 5.1: `//NP{//Adj->N}` (Q_3) is translated to the following SQL query. The subtree scope constraint requires that the nodes of tag N must be descendants of a node NP, and this is implemented by the conditions in bold.

```
select l l3, r r3, d d3, l2, r2, d2, l1, r1, d1
from T,
 ( select l l2, r r2, d d2, l1, r1, d1 from T,
   ( select l l1, r r1, d d1 from T
     where T.name = 'NP') T1
   where T.name = 'Adj' and T.l >= l1 and
       T.r <= r1 and T.d > d1) T2
where T.name = 'N' and T.l >= r2 and
  T.l >= l1 and T.r <= r1 and T.d > d1
```

■

To translate an LPath query with predicates to an SQL query, we use the techniques in [9]. When a predicate is met, we add the keyword EXISTS to the WHERE clause. The logical operators *and*, *or* in LPath predicates are directly mapped to keywords AND, OR in SQL. Operator NOT can be translated using NOT EXISTS in the SQL where clause. The key difference to the mapping proposed in [9] is that we also initialize the processing scope for expressions in the predicates to be the current scope.

As an optimization, rather than processing ^ and $ directly according to their definitions, we can evaluate these constraints efficiently according to the following observation. A node n is the rightmost descendant of node m if and only if n is a descendant of m, n and m share the same rightmost leaf descendant. According to the labeling scheme, n and m share the same rightmost leaf descendant if and only if $m.r = n.r$. Combining this with the strategy to evaluate the descendant axis, we can check $ by checking labels as follows. Let T' be the relation at the top of the scope stack which defines the current scope. ^A will be translated to conditions $T.name = A$, $T.l = T'.l$ and $T.d > T'.d$. Similarly, A$ will be translated to $T.name = A$, $T.r = T'.r$ and $T.d > T'.d$.

Example 5.2: The query `//VP{//NP$}` (Q_6) is translated to the following SQL query. The alignment $ is implemented by the condition in bold.

```
select l l2, r r2, d d2, l1, r1, d1 from T,
   ( select l l1, r r1, d d1 from T
     where T.name = 'VP') T1
 where T.name = 'NP' and T.l >= l1 and
       T.r = r1 and T.d > d1
```

■

6. CONCLUSIONS AND FUTURE WORK

We have addressed the problem of defining an expressive and efficient language for linguistic queries. Our language, LPath, extends the XPath language by introducing horizontal navigation primitives and subtree scoping, and expressing edge alignment naturally. We review each of these in turn. First, several new axes are proposed for horizontal navigation: immediate-following (->), immediate-following-sibling (=>), immediate-preceding (<-), and immediate-preceding-sibling (<=). These "horizontal" axes are not supported by XPath, even though their closures are supported. Once added, there is a natural symmetry between horizontal and the vertical axes (cf. Table 1). Second, subtree scoping is introduced using {}.

Finally, operators ^ and $ are used to express edge alignment naturally. When used in conjunction with {}, these force the specified node to be aligned to the left or right edge of the subtree.

For efficient evaluation of LPath queries, we have proposed a labeling scheme which supports both horizontal and vertical navigations. Based on the labeling scheme, we proposed a relational storage for linguistic tree data, and designed a query translator which converts LPath queries to SQL queries.

We believe this work has implications for XPath design and implementation beyond linguistics. First, we found that several important node navigations are not supported by XPath, presumably because these navigations are not requried in current applications. However, as XML is a standard data format representing a tree model, and XPath is its standard language, it is beneficial for XPath to support these navigations in order to support wider scientific applications. Furthermore, from a theoretical perspective, by including these primitive horizotal navigations in XPath, the set of XPath axes for horizontal navigation would be symmetric, just as the vertical navigation are, leading to an elegant inventory of axes.

The evaluation of LPath queries employs a novel labeling scheme which is also useful for XPath query processing. It is an interesting alternative to existing XPath query evaluation techniques.

In ongoing research, we are investigating the expressiveness of the language. For instance, we would like to support simple kinds of path closures (e.g. (->NP)*); as well as querying "overlapping trees" arising from multiple linguistic annotations over the same primary data. Finally, we plan to extend LPath with update operations, permitting local rearrangements of linguistic trees, and facilitating the curation of linguistic data.

7. ACKNOWLEDGMENTS

We would like to thank Val Tannen, Peter Buneman, and James Bailey for their valuable feedback on the work reported here. This research is sponsored by the National Science Foundation under Grant No. 0317826 *Querying Linguistic Databases.*

8. REFERENCES

[1] http://www.ldc.upenn.edu/Projects/QLDB/

[2] S. Bird, P. Buneman, and W.-C. Tan. Towards a query language for annotation graphs. In *Proceedings of the Second International Conference on Language Resources and Evaluation*, pages 807-814, 2000.

[3] S. Bird, M. Liberman. A formal framework for linguistic annotation *Speech Communication* 33, pages 23–60, 2001.

[4] S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, and Jerome Simeon XQuery 1.0: An XML Query Language `http://www.w3.org/TR/xquery/`.

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

[6] Y. Chen, S. Davidson, and Y. Zheng. BLAS: An Efficient XPath Processing System. In *Proceedings of SIGMOD*, pages 47–58, 2004.

[7] N. Chomsky. Formal properties of grammars. In E. Galanter D. Luce, R. R. Bush (eds), *Handbook of Mathematical Psychology*, volume 2, pages 323–418. New York: Wiley and Sons, 1963.

[8] J. Clark and S. DeRose. XML Path language (XPath), November 1999. `http://www.w3.org/TR/xpath`.

[9] D. DeHaan, D. Toman, M. Consens, and M. T. Ozsu. A comprehensive XQuery to SQL translation using dynamic interval encoding. In *Proceedings of SIGMOD*, pages 623–634, 2003.

[10] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. In *Proceedings of VLDB*, pages 95–106, 2002.

[11] D. Graff and S. Bird Many Uses, Many Annotations for Large Speech Corpora: Switchboard and TDT as Case Studies. Proceedings of the Second International Conference on Language Resources and Evaluation, pages 427–433, 2000.

[12] C. Lai and S. Bird. Querying and Updating Treebanks: A Critical Survey and Requirements Analysis Proceedings of the Australasian Language Technology Workshop, 2004.

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

[14] M. Marx. Conditional XPath, the first order complete XPath dialect. In *Proceedings of PODS*, pages 13–22, 2004.

[15] U. of Pennsylvania. The Penn Treebank Project, 1995. `http://www.cis.upenn.edu/~treebank/home.html`.

[16] P. Resnik and A. Elkiss. The Linguist's Search Engine: Getting Started Guide. Technical Report: LAMP-TR-108/CS-TR-4541/UMIACS-TR-2003-109, University of Maryland, College Park, November 2003. `http://lse.umiacs.umd.edu:8080/`.