

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

(Received 4 June 2005)

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 storing and querying linguistic data. 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. We provide a relational representation for trees, and define an SQL translation for queries. Experiments demonstrate that the query system is significantly faster than other linguistic tree query systems for a wide range of queries.

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 language technologies. These repositories may contain up to a billion words, along with annotations at the levels of phonetics, prosody, orthography, syntax, dialog, and gesture. The annotations are often hierarchical in nature, and are anchored to extents of text or speech.

Over a dozen linguistic query languages have been proposed, each with its own specialised interpreter for evaluating queries against a particular corpus of linguistic data.¹ Despite the considerable effort expended on developing these languages, little is known about their expressiveness and efficiency. As the size of the data grows and the analysis tasks become more complex, expressiveness and efficiency have become critical factors.

In this paper we present 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. In section 2 we describe the ordered tree data model, introduce a working example, and list a variety of query requirements. Next, in section 3, we propose an expressive and intuitive linguistic query language by extending

¹ Examples of such linguistic query languages include (Cassidy and Harrington, 2001; Rohde, 2001; Randall, 2000; Bird et al., 2000). We refer the reader to (Lai and Bird, 2004) for a critical survey of tree query languages.

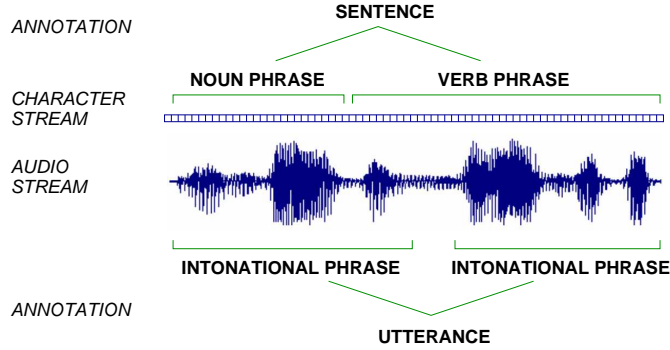


Fig. 1. Linguistic Annotation: Structured Coding of Extents of Time-Series Data (e.g. Character Data, Audio Data)

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 labelling scheme which supports efficient horizontal and vertical tree navigations. Given this language and labelling 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. Experiments, reported in section 6, show that the proposed approach performs well on various data and query sets. Section 7 concludes the paper with a summary and a discussion of future research.

2 Querying Linguistic Trees

Linguistic databases consist of time-series data coupled with hierarchical annotations (Bird and Liberman, 2001). The time-series data represents an external linguistic artefact, and takes the form of a text or recording. The relationship between the primary data and its annotations is shown schematically in Figure 1. Observe that the nodes of the tree are ordered, by virtue of the linear ordering of the time-series data.

A common data model for linguistic annotations is a labelled, ordered tree. For example, Figure 2(a) shows the tree representation of a parsed sentence. A natural candidate for representing this is XML, as shown in Figure 2(b).

Much data-intensive language processing involves searching and collating this 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)

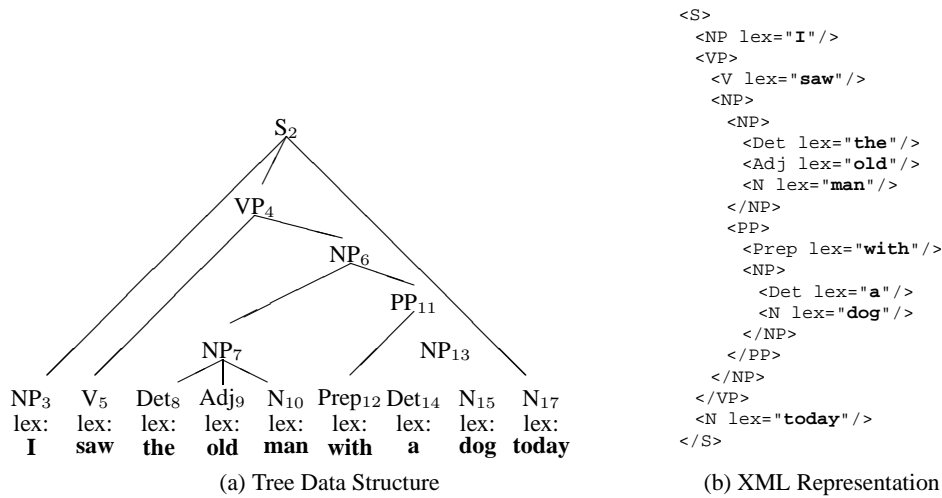


Fig. 2. Linguistic Tree Representations

- 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})

The rest of this section discusses three key features of linguistic queries: node navigation, subtree scoping, and edge alignment.

Node navigation. Linguistic trees need to be navigated in both vertical and horizontal directions. To process query 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 any 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.

Query Q_2 searches for noun phrases that immediately follow a verb. 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 productions in reverse to a sentence in order to get sequences, or so-called ‘proper analyses’ (Chomsky, 1963). For example, Figure 3(b) shows some proper analyses of the sentence *I saw the old man with*

S	→	NP VP (NP)	<i>I saw the old man PP₁₁ today</i>
VP	→	V NP (NP)	<i>I V₅ Det₈ Adj₉ N₁₀ PP₁₁ today</i>
NP	→	NP PP	<i>I V₅ NP₇ PP₁₁ today</i>
NP	→	Det Adj* N	<i>I V₅ NP₆ today</i>
PP	→	Prep NP	<i>I VP today</i>

(a) CFG Productions

(b) Some Proper Analyses

Fig. 3. CFG and its Proper Analyses

a dog today with respect to the grammar in Figure 3(a). These have been co-indexed with the tree nodes in Figure 2.

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 result of query Q_2 .

Next consider query Q_3 , which involves the ‘follows’ 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 .

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. We 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 .

There is a fourth feature of linguistic queries, exemplified in query Q_7 . As we shall see, this feature is nothing other than the combination of the three we have already seen. Here, we have the situation where 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 V₅ NP₇ PP₁₁ today* in Figure 3(b), we can apply the production rule $NP \rightarrow NP PP$ in reverse to get another proper analysis *I V₅ NP₆ today*. Next, applying the production rule $VP \rightarrow V NP$ in reverse, we get *I VP₄ today*. Thus, 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 .

In the next section we propose a new linguistically-motivated tree language which supports these expressive requirements.

3 LPath: A path language for linguistic trees

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. In this section we propose a new path language for linguistic trees, balancing expressiveness and efficiency. We chose XPath (Clark and DeRose, 1999) as the basis of our language given the focus on locating nodes in a tree instead of transforming and constructing trees, a task served by the XQuery language (Boag et al., 2004). 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 (Gottlob et al., 2002; Marx, 2004). Also various evaluation and optimisation techniques for XPath have been proposed (Bruno et al., 2002; Chen et al., 2004; Li and Moon, 2001).

We begin by presenting the navigation ‘axes’ of LPath. Next, we define the grammar of LPath 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 navigations to retrieve children and parents, and their transitive closures to retrieve descendants and ancestors. These axes are well-known and already defined in XPath, and so we do not discuss them further.

LPath supports sibling navigation to retrieve the immediately following sibling, immediately preceding sibling, and their transitive closures to retrieve following siblings and preceding siblings (Marx, 2004). 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 more generally useful for ordered trees, independently of context free grammar. We define it formally as follows:

Definition: Let $w_1 \dots w_k$ be an ordered sequence of terminals and let T be an annotation tree built over the sequence. The set of nodes retrieved by applying the *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:

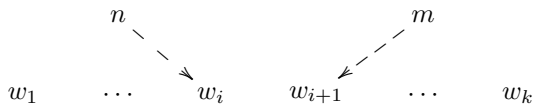


Table 1. *LPath Navigation Axes*

		Vertical		
/	child		\	parent
//	descendant (/ ⁺)		\\	ancestor (\\ ⁺)
	descendant-or-self (/ [*])			ancestor-or-self (\\ [*])
		Horizontal		
->	immediate-following		<-	immediate-preceding
-->	following (-> ⁺)		<--	preceding (<- ⁺)
	following-or-self (-> [*])			preceding-or-self (<- [*])
		Sibling		
=>	immediate-following-sibling		<=	immediate-preceding-sibling
==>	following-sibling (=> ⁺)		<==	preceding-sibling (<= ⁺)
	following-sibling-or-self (=> [*])			preceding-sibling-or-self (<= [*])
		Other		
.	self		@	attribute

We define the inverse axis, *immediately preceding* in the obvious way. We further define the *following* (resp. *preceding*) axis as the transitive closure of the immediately following (resp. preceding) axis.²

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.³

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.⁴ 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’.

² Note that these transitive closures are equivalent to the *primitive* transitive relations already defined in XPath. Here we have generalized Marx’s approach to the sibling axis, applying it to the horizontal axis.

³ 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.

⁴ Instead of using $*$ as a wildcard to match any tag name as defined in XPath, we use $_$ as wildcard and $*$ to denote transitive closure.

```

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)

Fig. 4. The Grammar of LPath

Subtree scoping. We introduce braces into the language to permit subtree scopes to be expressed. 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 subtree rooted at the node matching n . For example, consider query 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: $//VP\{ /V-->N\}$. Now consider the 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 .⁵

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 \wedge 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: $\wedge A = A[not<--_]$; $A\$ = A[not-->_]$. (A more efficient evaluation for \wedge and $\$$ is given in section 5.) Accordingly, Q_5 can be expressed as: $//VP\{ /NP\$$.

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 2.

- 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`

⁵ 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 readers to section 5 for more details.

- 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. An interesting question is whether we should express the extended navigation axes of LPath as user defined functions in XPath and 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 common in linguistic queries, and it is crucial that horizontal navigation is 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).

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 `^` 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 Labelling Scheme

One important class of XPath query engines employ labelling schemes that encode a node by its position. These methods have been shown to be very efficient (Li and Moon, 2001; Bruno et al., 2002; Chen et al., 2004). There are at least three such labelling schemes in the literature. One is proposed in (Dietz, 1982), using triples of the form $(pre, post, depth)$, where $pre, post$ are the positions of the node in a pre-order and post-order tree traversal. Another is adapted from inverted lists in information retrieval (Salton and McGill, 1983; Zhang et al., 2001; DeHaan et al., 2003), and uses triples of the form $(start, end, depth)$, where $start, end$ are the textual positions of the start and end tag of a node. The third scheme is based on the size of the node enclosed by the start tag and end tag, and uses triples of the form $(pre, size, height)$ or $(start, size, height)$ (Li and Moon, 2001). There is also work on updateable labels (Dietz and Sleator, 1987; Silberstein et al., 2005). While these labelling schemes enable us to determine the vertical navigation relationships between two nodes efficiently by checking the containment relationship of their labels, they do not address the problem of evaluating the horizontal navigations required in linguistic queries.

Therefore, we propose a new labelling scheme to capture the structure of linguistic trees, based on the chart data structure from computational linguistics (Gazdar and Mellish, 1989, 179ff) and the annotation graph model (Bird et al., 2000; Bird and Liberman, 2001; Ma et al., 2002). It will be used to facilitate detection of linguistically-motivated relationships between nodes.

The labelling scheme assigns each node a tuple: $\langle left, right, depth, id, pid, name \rangle$, abbreviated to $\langle l, r, d, id, pid, name \rangle$, in the following fashion:

1. Let n be the leftmost leaf element; assign $n.l = 1$.
2. Let n be a leaf element; assign $n.r = n.l + 1$.
3. Let m and n be consecutive leaf elements where m is on the left; assign $m.r = n.l$.
4. Let n be a non-terminal element which has a sequence of leaf descendants in order: m_1, \dots, m_k ; 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 $\langle l, r, d, id, pid \rangle$ 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 .

In Table 2 we show how we can determine the LPath axis relationship of any two nodes simply by inspecting their labels.⁶

⁶ Extensions to reflexive versions of the axes are easy and are omitted here. For example, $descendant\text{-or-self}(m, n) = m.l \geq n.l, m.r \leq n.r, m.d \geq n.d$

Table 2. Axes and Their Corresponding Label Comparisons

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

Table 3. Relational Representation T

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
...						

As an illustration of this scheme, consider again the tree in Figure 2. The corresponding labels are given in Table 3, where the id attribute in the table T corresponds 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 $S_2.d < NP_6.d$ 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 3, the two key features of a good query 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 focus on efficiency.

5.1 Query System Requirements

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, annotators correct the mechanically parsed data, supervisors implement quality controls, and external experts supply highly specialised 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 contains substantial tabular data in addition to trees, for 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, we 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') which must be found using the pronunciation table since it is not obvious from spelling (e.g. *face* contains *s*, and *nation* contains *sh*). Furthermore, we may want to work within a sub-collection 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 (Resnik and Elkiss, 2003). For such applications, the data will not fit in main memory. Thus the system must support efficient queries over data stored on disk.

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

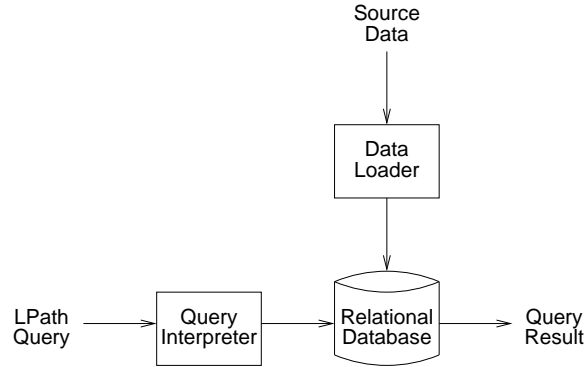


Fig. 5. The System Architecture

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 labelling scheme presented in section 4. It is composed of three modules: a data loader, a query translator, and a relational database. 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 labelling scheme. We use a commercial relational database as the query engine. The architecture of our LPath query processing system is presented in Figure 5.

5.3 System Description

The data loader generates a tuple $\langle \text{left}, \text{right}, \text{depth}, \text{id}, \text{pid}, \text{name}, \text{value} \rangle$ for each node in a tree, and stores it into a relational table. For example, Table 3 shows part of the relation generated for the sample annotation tree in Figure 2. Indexes are built on the table to facilitate searches.

The query translator converts an LPath query to an SQL query based on the labelling 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. We then 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 $n.d > m.d$. When the corresponding $\}$ is met, we pop the current scope from the stack.

Example: $//NP\{ //Adj \rightarrow N\}$ (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 (DeHaan et al., 2003). 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 (DeHaan et al., 2003) is that we also initialise the processing scope for expressions in the predicates to be the current scope.

Rather than processing \wedge and \S directly according to their definitions, we evaluate these constraints more efficiently as follows. A node n is the rightmost descendant of node m if and only if n is a descendant of m , and if n and m have the same rightmost leaf descendant. According to the labelling scheme, n and m share the same rightmost leaf descendant if and only if $m.r = n.r$. Now suppose T' is the relation at the top of the scope stack which defines the current scope. Then $\wedge A$ will be translated to conditions $T.name = A$, $T.l = T'.l$ and $T.d > T'.d$. Similarly, $A\S$ will be translated to $T.name = A$, $T.r = T'.r$ and $T.d > T'.d$.

Example: The query $//VP\{ //NP\S \}$ (Q_6) is translated to the following SQL query. The alignment \S 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 Experimental Results

We have implemented the LPath compiler in C++, and it is available from our website.⁷ The labelled form of linguistic trees are stored in relational databases using Oracle 10g personal version, with schema $\{tid, left, right, depth, id, pid, name, value\}$. The attribute tid is used to distinguish different trees, and $value$ is used to record data values. The relation is clustered by $\{name, tid, left, right, depth, id, pid\}$, and has indexes on $\{tid, value, id\}$, $\{value, tid, id\}$ and $\{tid, id, left, right, depth, pid\}$. We use Yacc to generate a parser to translate an LPath query to a SQL query. In this section we report on experimental evaluation and performance comparison with other linguistic tree query systems.

⁷ <http://www ldc.upenn.edu/Projects/QLDB/>

(a) Test Data Sets			(b) Ten Most-Frequent Tags				
	WSJ	SWB	Tag	WSJ Frequency	Tag	SWB Frequency	
File Size	35983kB	35880kB	1	NP	292430	-DFL-	193708
Tree Nodes	3484899	3972148	2	VP	180405	VP	185259
Unique Tags	1274	715	3	NN	163935	NP-SBJ	135867
Maximum Depth	36	36	4	IN	121903	.	135753
			5	NNP	114053	,	133528
			6	S	107570	S	132336
			7	DT	101190	NP	129804
			8	NP-SBJ	95072	PRP	114332
			9	-NONE-	79247	NN	76390
			10	JJ	75266	RB	73477

Fig. 6. Test Data

6.1 Experimental Setup

The experiments were performed on a 2GHz Pentium 4 machine, with 512M memory and one 7200rpm hard disk. All experiments were repeated 7 times independently, and the average processing time for the Oracle query evaluation was measured, disregarding the maximum and minimum values.

6.1.1 Systems

We compare the performance of the LPath query engine with two popular linguistic query language implementations, TGrep2 (Rohde, 2001) and CorpusSearch (Randall, 2000). TGrep2 is a query engine for finding structures in a database of linguistic trees. The most common application of TGrep2 is extracting data from the Penn Treebank corpus of parsed sentences. Query execution uses a binary file representation of the data, including an index on the words in the trees. CorpusSearch was also developed to query corpora annotated in the Penn Treebank style. (The expressiveness of CorpusSearch with respect to TGrep2 is unknown.) We also present the performance of an XPath engine using a popular XML labelling scheme for comparison (Salton and McGill, 1983; Zhang et al., 2001; DeHaan et al., 2003).

6.1.2 Data Sets

The two data sets used were the Wall Street Journal section of the Penn Treebank corpus, and the Switchboard corpus:

Wall Street Journal: This data set was created by the Penn Treebank Project (Marcus et al., 1993), in which 2,499 stories were selected from a three year Wall Street Journal collection of 98,732 stories for syntactic annotation (WSJ).

	LPath Query	Size of WSJ Result	Size of SWB result
Q_1	//S[//_[@lex=saw]]	153	339
Q_2	//VB->NP	23618	16557
Q_3	//VP/VB-->NN	63857	32386
Q_4	//VP{/VB-->NN}	46116	25305
Q_5	//VP{/NP\$}	29923	22554
Q_6	//VP{/NP\$}	215104	112159
Q_7	//VP[{/^VB->NP->PP\$}]	2831	1963
Q_8	//S[/NP/ADJP]	7832	2900
Q_9	//NP[not(//JJ)]	211392	109311
Q_{10}	//NP[->PP[/IN[@lex=of]]=>VP]	192	31
Q_{11}	//S[{//_[@lex=what] -->_[@lex=building]}]	2	5
Q_{12}	//_[@lex=rapprochement]	1	0
Q_{13}	//_[@lex=1929]	14	0
Q_{14}	//ADVP-LOC-CLR	60	0
Q_{15}	//WHPP	87	20
Q_{16}	//RRC/PP-TMP	8	3
Q_{17}	//UCP-PRD/ADJP-PRD	17	4
Q_{18}	//NP/NP/NP/NP/NP	254	12
Q_{19}	//VP/VP/VP	8769	6093
Q_{20}	//PP=>SBAR	640	651
Q_{21}	//ADVP=>ADJP	15	37
Q_{22}	//NP=>NP=>NP	7	7
Q_{23}	//VP=>VP	20	72

Fig. 7. Test Queries

Switchboard: Switchboard (Greenberg, 1996) is a collection of about 2,400 two-sided telephone conversations among 543 speakers from all areas of the United States. This data set includes parsed text of 650 conversations from the Switchboard transcripts (SWB).

Characteristics of these data sets are presented in Figure 6(a). *File Size* is the disk space required for the uncompressed ASCII representation of the linguistic trees. *Tree Nodes* is the number of nodes in the given file, including element and attribute nodes. *Unique Tags* is the number of distinct tags. *Maximum Depth* is the length of the longest root-to-leaf path in the tree representation. We list the ten most frequent tags appearing in each data set in Figure 6(b).

6.1.3 Query sets

Figure 7 shows the 23 queries tested on each data set. In addition to the queries discussed earlier (Q_1 to Q_7), we added queries with value tests in predicates (Q_1 , Q_{10} to Q_{13}), and

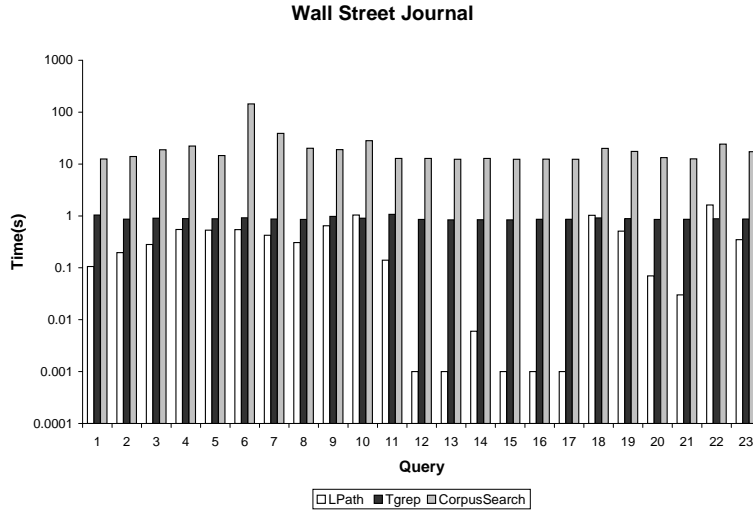


Fig. 8. Query Execution Time on Wall Street Journal Data Set

queries containing sibling axis traversals (Q_{20} to Q_{23}). We include both queries with high selectivity (Q_{11} , Q_{12} , Q_{16} and Q_{17}) and with low selectivity (Q_6 and Q_9). Some of the queries use different tags to the ones presented in section 2 in order to match corresponding tags in data sets. 11 out of these 23 queries are expressible in XPath (for example, Q_8). In the experiment, we return just the result size.

6.2 Query Processing Time

Figures 8 and 9 present the query execution time in log scale for the three systems for the WSJ and SWB data sets, respectively.⁸ For the WSJ data set, the LPath query engine is the fastest except for queries Q_{10} , Q_{18} and Q_{22} . In each of these three queries, low selectivity tags appear. Q_{10} contains the most frequent tag NP , the second most frequent tag VP , and the fourth most frequent tag IN ; Q_{18} contains NP 5 times in a row vertically; and Q_{22} contains NP 3 times horizontally. When such low selectivity tags appear in a query, a lot of disk accesses are needed and the size of the intermediate results is big. On the other hand, when the tags appearing in a query have high selectivity, the LPath query engine works well by leveraging the indexes, e.g. Q_{16} and Q_{17} . Furthermore, the LPath query engine performs well for queries containing high-selectivity value predicates, as they effectively reduce the size of intermediate results for joins, e.g. Q_{11} and Q_{12} . For the SWB data set, the LPath query engine is the fastest for all queries since the frequency of the queried tags is much lower than in WSJ. In particular, the tags appearing in the query set which are highly

⁸ In the case of CorpusSearch we had to revise queries Q_2 and Q_7 , since they could not be translated exactly.

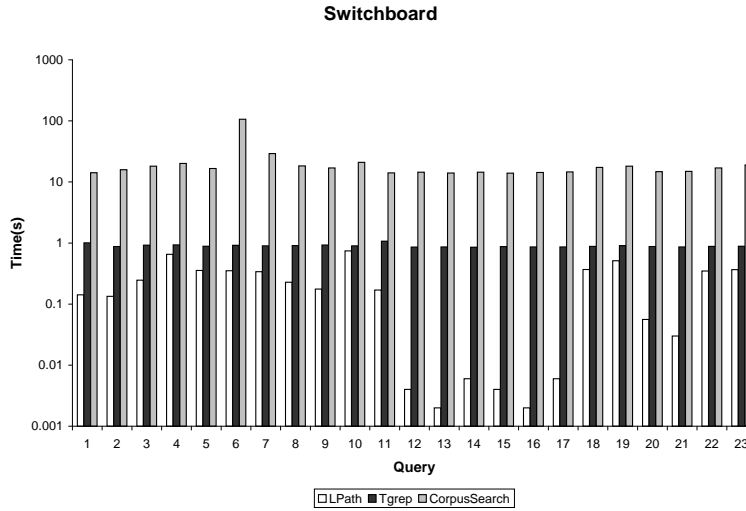


Fig. 9. Query Execution Time on Switchboard Data Set

frequent in the WSJ data set generally occur with much lower frequency in the SWB data set.

6.3 Scalability of Query Processing Time

To test the scalability of these systems as the data size increases, we replicated the *WSJ* data set between 0.5 and 4 times to get experimental data sets of varying sizes. Figure 10 reports the processing time on data of increasing sizes for a representative sample of queries having different types. The performance of other queries is similar and is omitted. As we can see, LPath scales well.

6.4 Labelling Scheme

We compared the labelling scheme in the LPath query engine with a well-known labelling scheme designed for evaluating XPath queries (Salton and McGill, 1983; Zhang et al., 2001; DeHaan et al., 2003), which is referred to here as the XPath interval labelling scheme. This scheme uses textual positions of the start and end tags rather than `left` and `right` as used in our labelling scheme. The XPath interval labelling scheme was proposed to efficiently evaluate the descendant axis and the child axis by testing label containment. To compare performance, we set other components of both labelling schemes to be the same. Figure 11(a,b) reports the query execution time on the *WSJ* and *SWB* data sets. Eleven queries in the query set can be expressed using XPath and therefore supported by the XPath interval labelling scheme. As we can see, the performance of these two labelling schemes is almost the same. Thus, the LPath labelling scheme supports more queries without degrading performance relative to XPath query evaluation.

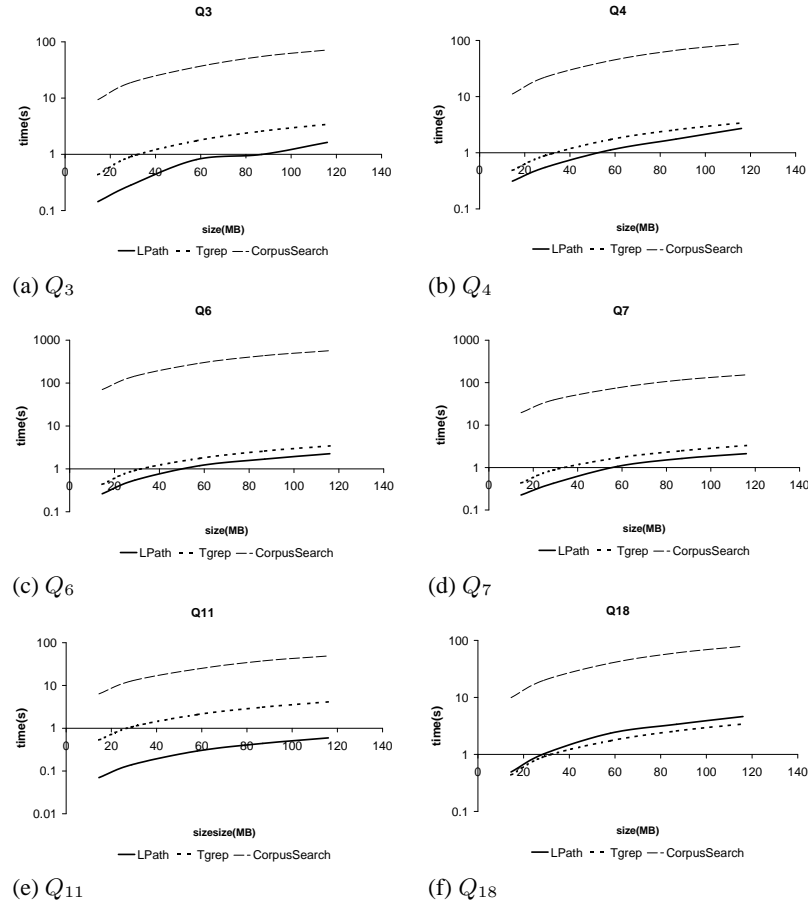


Fig. 10. Query Execution Time as WSJ Data Size Increases

7 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 three devices: horizontal navigation; subtree scoping, and edge alignment. 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. Thus, we have filled an important gap in XPath. Second, subtree scoping is introduced. If $P\{Q\}$ is an LPath expression, then a complex expression Q cannot return nodes outside the subtree dominated by P . Finally, operators \wedge 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.

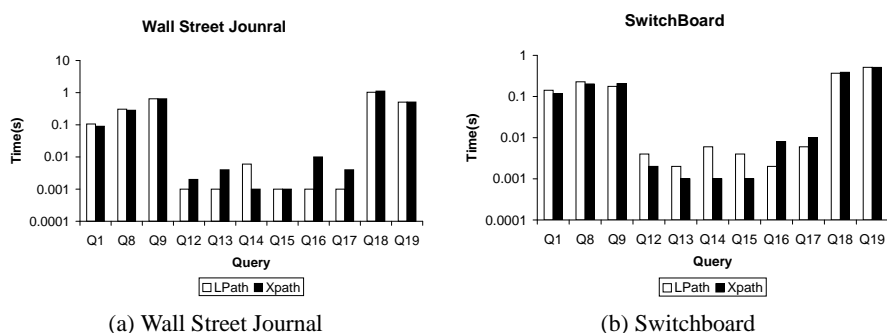


Fig. 11. Query Execution Time

For efficient evaluation of LPath queries, we have proposed a labelling scheme which supports both horizontal and vertical navigations. Based on the labelling scheme, we proposed a relational storage model for linguistic tree data based on the chart data structure, 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 required 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 horizontal 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 labelling scheme which is also useful for XPath query processing. As shown in section 6, an LPath query engine has the same performance as an XPath query engine, but supports more queries. 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 develop an extended language LPath+ containing simple kinds of path closures (e.g. $(\rightarrow_{NP})^*$), and investigate the formal properties of this language with respect to Conditional XPath and Propositional Dynamic Logic (Marx, 2004; Kracht, 2003; Lai, 2005). We would also like to extend the data model to permit ‘overlapping trees’ that arise 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.

8 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*.

References

- Bird, S., Buneman, P., and Tan, W.-C. (2000). Towards a query language for annotation graphs. In *Proceedings of the Second International Conference on Language Resources and Evaluation*, pages 807–814. Paris: ELRA. <http://arXiv.org/abs/cs/0007023>.
- Bird, S. and Liberman, M. (2001). A formal framework for linguistic annotation. *Speech Communication*, 33:23–60.
- Boag, S., Chamberlin, D., Fernandez, M., Florescu, D., Robie, J., and Simeon, J. (2004). *XQuery 1.0: An XML Query Language*. W3C. <http://www.w3.org/TR/xquery/>.
- Bruno, N., Koudas, N., and Srivastava, D. (2002). Holistic twig joins: Optimal XML pattern matching. In *Proceedings of SIGMOD*, pages 310–321.
- Cassidy, S. and Harrington, J. (2001). Multi-level annotation of speech: an overview of the Emu Speech Database Management System. *Speech Communication*, 33:61–77.
- Chen, Y., Davidson, S., and Zheng, Y. (2004). Blas: An efficient XPath processing system. In *Proceedings of SIGMOD*, pages 47–58.
- Chomsky, N. (1963). Formal properties of grammars. In Galanter, E., Luce, D., and Bush, R. R., editors, *Handbook of Mathematical Psychology*, volume 2, pages 323–418. New York: Wiley and Sons.
- Clark, J. and DeRose, S. (1999). *XML Path language (XPath)*. W3C. <http://www.w3.org/TR/xpath>.
- DeHaan, D., Toman, D., Consens, M., , and Ozsu, M. T. (2003). A comprehensive XQuery to SQL translation using dynamic interval encoding. In *Proceedings of SIGMOD*, pages 623–634.
- Dietz, P. F. (1982). Maintaining order in a linked list. In *Proceedings of the 14th Annual ACM Symposium on the Theory of Computing*, pages 122–127.
- Dietz, P. F. and Sleator, D. D. (1987). Two algorithms for maintaining order in a list. In *Proceedings of the 19th Annual ACM Symposium on the Theory of Computing*, pages 365–372.
- Gazdar, G. and Mellish, C. (1989). *Natural Language Processing in Prolog: An Introduction to Computational Linguistics*. Addison-Wesley.
- Gottlob, G., Koch, C., and Pichler, R. (2002). Efficient algorithms for processing XPath queries. In *Proceedings of the 28th Conference on Very Large Data Bases*, pages 95–106.
- Greenberg, S. (1996). The switchboard transcription project. LVCSR Summer Research Workshop, Johns Hopkins University.
- Kracht, M. (2003). *The Mathematics of Language*, volume 63 of *Studies in Generative Grammar*. Mouton de Gruyter, Berlin.
- Lai, C. (2005). Querying treebanks. Master’s thesis, Department of Computer Science, University of Melbourne.
- Lai, C. and Bird, S. (2004). Querying and updating treebanks: A critical survey and requirements analysis. In *Proceedings of the Australasian Language Technology Workshop*, pages 139–146. <http://eprints.unimelb.edu.au/archive/00000774/>.
- Li, Q. and Moon, B. (2001). Indexing and querying XML data for regular path expressions. *The VLDB Journal*, 10:361–370.

- Ma, X., Lee, H., Bird, S., and Maeda, K. (2002). Models and tools for collaborative annotation. In *Proceedings of the Third International Conference on Language Resources and Evaluation*, pages 2066–2073. Paris: ELRA. <http://arXiv.org/abs/cs/0204004>.
- Marcus, M. P., Santorini, B., and Marcinkiewicz, M. A. (1993). Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–30. <http://www.cis.upenn.edu/~treebank/home.html>.
- Marx, M. (2004). Conditional XPath, the first order complete xpath dialect. In *Proceedings of PODS*, pages 13–22.
- Randall, B. (2000). Corpusearch. <http://www.cis.upenn.edu/~brandall/CSStuff/CSManual/Contents.html>.
- Resnik, P. and Elkiss, A. (2003). 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. <http://lse.umiacs.umd.edu:8080/>.
- Rohde, D. (2001). Tgrep2 user manual. <http://citeseer.ist.psu.edu/569487.html>.
- Salton, G. and McGill, M. (1983). *Introduction to Modern Information Retrieval*. McGraw-Hill.
- Silberstein, A., He, H., Yi, K., and Yang, J. (2005). BOXes: Efficient maintenance of order-based labeling for dynamic xml data. In *Proceedings of the 21st International Conference on Data Engineering*, pages 285–296.
- Zhang, C., J. F. Naughton, D. J. D., Luo, Q., and Lohman, G. M. (2001). On supporting containment queries in relational database management systems. In *Proceedings of SIGMOD*, pages 425–436.