Towards an Alternative Implementation of NXT's Query Language via XQuery

Neil Mayo, Jonathan Kilgour, and Jean Carletta
University of Edinburgh

Abstract

The NITE Query Language (NQL) has been used successfully for analysis of a number of heavily cross-annotated data sets, and users especially value its elegance and flexibility. However, when using the current implementation, many of the more complicated queries that users have formulated must be run in batch mode. For a re-implementation, we require the query processor to be capable of handling large amounts of data at once, and work quickly enough for on-line data analysis even when used on complete corpora. Early results suggest that the most promising implementation strategy is one that involves the use of XQuery on a multiple file data representation that uses the structure of individual XML files to mirror tree structures in the data, with redundancy where a data node has multiple parents in the underlying data object model.

1 Introduction

Computational linguistics increasingly requires data sets which have been annotated for many different phenomena which relate independently to the base text or set of signals, segmenting the data in different, conflicting ways. The NITE XML Toolkit, or NXT (Carletta et al., 2003), has been used successfully on a range of text, spoken language, and multimodal corpora to provide and work with data of this character. Because the original motivation behind it was to make up for a dearth of tools that could be used to hand-annotate and display such data, the initial implementation of data search was required to work well on one data observation at a time — that is, one text, dialogue, or other language event — and to be usable but slow on multiple observations. However, the clear and flexible design of NXT's query language, NQL (Heid et al., 2004; Carletta et al., in press), makes it attractive for larger-scale data analysis, and a number of users have up-translated existing data for the express purpose of improving their search options.

We are now in the process of devising a strategy for re-implementing the NQL processor to serve the needs of this class of user better. In this paper, we describe our requirements for the new implementation, outline the various implementation options that we have, and give early results suggesting how well they meet our requirements. NQL is arguably the most mature of the current special-purpose facilities for searching data sets where the data is not structured as a single tree, and therefore experiences with implementing it are likely to provide lessons for search facilities that are still to come.

2 NXT and the NITE Query Language

NXT is designed specifically for data sets with multiple kinds of annotation. It requires data to be represented as a set of XML files related to each other using stand-off links, with a "metadata" file that provides two things: a catalogue of files containing the audio or video signals used to capture an observation together with the annotations that describe them, and a specification of the corpus design that describes the annotations and how they relate to each other and to signal. Text corpora are treated as signal-less, with the text as a base level of "annotation" to which other annotations can point. Given data of this type, NXT provides Java libraries for data modelling and search as well as for building graphical user interfaces that can be used to display annotations in synchrony with the signals. NXT also comes with a number of finished GUIs for common tasks, some of which are specific to existing data sets and some of which are configurable to new corpus designs. NXT supports data exploration using a search GUI, callable from any tool, that will run an NQL query and highlight results on the tool's display. Data search is then usually done using one of a battery of command line utilities, that, for instance, count results matching a given query or provide tab-delimited output describing the query matches.

Because the data model and query language for a tool are critical to our implementation choices, we briefly describe them, as well as the current NQL implementation.

2.1 NXT's data handling

In NXT, annotations are described by types and attribute value pairs, and can relate to a synchronized set of signals via start and end times, to representations of the external environment, and to each other. Annotations can describe the behaviour of a single 'agent', if more than one participant is involved for the genre being coded, or they can describe the entire language event; the latter possibility is used for annotating written text as well as for interaction within a pair or group of agents. The data model ties the annotations together into a multi-rooted tree structure that is characterized by both temporal and structural orderings. Additional relationships can be overlaid on this structure using pointers that carry no ordering implications. The same basic data objects that are used for annotations are also used to build sets of objects that represent referents from the real world, to structure type ontologies, and to provide access to other external resources stored in XML, resulting in a rich and flexible system.

Data is stored in a "stand-off" XML representation that uses the XML structure of individual files to mirror the most prominent trees in the data itself, forming 'codings' of related annotations, and pointers between files (represented by `XLinks`) for other relationships. For example, the markup

```
<nt nite:start="1.8" nite:end="2.3"
cat="NP" nite:id="s1_506" wc="2">
<nite:child href="a1.words.xml#id(s1_6)"/>
<nite:child href="a1.words.xml#id(s1_7)"/>
</nt>
```

represents a noun phrase in a syntactic tree, pointing to two words in a different file which constitute the content of that syntactic structure.

This has the useful properties of allowing corpus subsets to be assembled as needed; making it easy to import annotations without perturbing the rest of the data set; and keeping each individual file simple to use in external processing. For instance, the words for a single speaker can be stored in a single file that contains no other data, making it easy to pass through to a part-of-speech tagger. NXT itself provides transparent corpus-wide access to the data, and so tool users need not understand how the data is stored. A 'metadata' file defines the structures of the individual files and the relationships among them, as well as detailing where to find the data and signals on the file system.

2.2 NXT's query language

Search is conducted in NXT by means of a dedicated query language, NQL. A simple query finds n-tuples of data objects (annotations and objects) that satisfy certain conditions. The query expression consists of two parts, separated by a colon (:). In the first part, variables representing the data objects are declared. These either match all data objects (e.g. '($x)' for a variable named '$x') or are constrained to draw matches from a designated set of simple types (e.g. '($w word || sil)', matching data objects of the simple types 'word' or 'sil'). The second part of the query expression specifies a set of conditions that are required to hold for a match, which are combined using negation (logical not, '!'), conjunction (logical and, '&&'), and disjunction (logical or, '||'). Queries return a structure that for each match lists the variables and references the data objects they matched.

NQL has operators that allow match conditions to be expressed for each of the essential properties of a data object such as its identity, its attribute-value pairs, its textual content, its timing, and its relationships via the two types of structural links (child and pointer). The attribute and textual content tests include the ability to match against either the existence or the value of the attribute.

Attribute values or textual content can be tested against explicit values or values of other attributes, using equality, inequality, and the usual ordering tests (conventionally represented as $<$, $<=$, $>=$, and $>$). String values can also be tested against regular expressions.

The temporal operators include the ability to test whether a data object has timing information, and to compare the start or end time with a given point in time. The query language also has operators to test for some common timing relationships between two data objects, such as overlap.

The structural operators test for dominance, precedence, and pointer relationships. Precedence can be tested against all of the orderings in the overlapping annotations.

In addition, identity tests can be used to avoid matches where different variables point to the same data object. It is also possible in NQL to 'bind' variables within the query using existential ('exists') and universal ('forall') quantifiers in the variable declarations (which have the same meaning as in first-order logic). Such bound variables are not returned in the query result.

NXT also supports the sequencing of queries into a 'complex' query using a double colon (::) operator. The results for a complex query are returned not as a flat list but as a tree structure. For example, in a corpus of timed words from two speakers, A and B,

```
  ($wa word):($wa@agent = "A")::
($wb word):($wb@agent="B") && ($wa
overlaps.with $wb)
```

will return a tree showing word overlaps; underneath each top level node, representing an overlapping word from speaker A, will be a set of nodes representing the words from speaker B that overlap that word of speaker A.

2.3 Comparison to other search facilitiies

The kinds of properties that linguists wish to use in searching language data are cumbersome to express in general-purpose query languages. For this reason, there are a number of other query languages designed specifically for language corpora, some of which are supported by implementation. LPath (Bird et al., 2006) and tgrep2 (Rohde, nd) assume the data forms one ordered tree. TigerSearch (Tiger Project, nd) is primarily for single trees, but does allow some out-of-tree relationships; the data model includes "secondary edges" that link a node to an additional parent and that can be labelled, with query language operators that will test for the presence or absence of such an edge, with or without a specific label. ATLAS (National Institute of Standards and Technology, 2000) intends a query language over richer structures, but the structures and query language are still under development.

3 Requirements

We already have a successful NQL implementation as part of NXT, NXT Search. However, as always, there are a number of things that could be improved about it. We are considering a re-implementation with the following aims in mind:

Faster query execution. Although many queries run quite quickly in NXT Search, more complicated queries can take long enough to execute on a large corpus that they have to be scheduled overnight. This is partially due to the approach of checking every possible combination of the variables declared in the query, resulting in a large search space for some queries. Our aim is to have the vast majority of queries that exploit NXT's multi-rooted tree structure run quickly enough on single observations that users will be happy to run them in an interactive GUI environment.

The ability to load more data. NXT loads data into a structure that is 5-7 times the size of the data on disk. A smaller memory representation would allow larger data sets to be loaded for querying. Because it has a "lazy" implementation that only loads annotations when they are required, the current performance is sufficient for many purposes, as this allows all of the annotations relating a single observation to be loaded unless the observation is both long and very heavily annotated. It is too limited (a) when the user requires a query to relate annotations drawn from different observations, for instance, as a convenience when working on sparse phenomena, or when working on multiple-document applications such as the extraction of named entities from newspaper articles; (b) for queries that draw on very many kinds of annotation all at the same time on longer observations; and (c) when the user is in an interactive environment such as a GUI using a wide range of queries on different phenomena. In the last case, our goal could be achieved by memory management that throws loaded data away instead of increasing the loading capacity.

4 XQuery as a Basis for Re-implementation

XQuery (Boag et al., 2005), currently a W3C Candidate Recommendation, is a Turing-complete functional query/programming language designed for querying (sets of) XML documents. It subsumes XPath, which is “a language for addressing parts of an XML document” (Clark and DeRose, 1999). XPath supports the navigation, selection and extraction of fragments of XML documents, by the specification of ‘paths’ through the XML hierarchy. XQuery queries can include a mixture of XML, XPath expressions, and function calls; and also FLWOR expressions, which provide various programmatical constructs such as for, let, $where$, $orderby$ and $return$ keywords for looping and variable assignment. XQuery is designed to make efficient use of the inherent structure of XML to calculate the results of a query.

XQuery thus appears a natural choice for querying XML of the sort over which NQL operates. Although the axes exposed in XPath allow comprehensive navigation around tree structures, NXT’s object model allows individual nodes to draw multiple parents from different trees that make up the data; expressing navigation over this multi-tree structure can be cumbersome in XPath alone. XQuery allows us to combine fragments of XML, selected by XPath, in meaningful ways to construct the results of a given query.

There are other possible implementation options that would not use XQuery. The first of these would use extensions to the standard XPath axes to query concurrent markup, as has been demonstrated by (Iacob and Dekhtyar, 2005). We have not yet investigated this option.

The second is to come up with an indexing scheme that allows us to recast the data as a relational database, the approach taken in LPath (Bird et al., 2006). We chose not to explore this option. It is not difficult to design a relational database to match a particular NXT corpus as long as editing is not enabled. However, a key part of NXT’s data model permits annotations to descend recursively through different layers of the same set of data types, in order to make it easy to represent things like syntax trees. This makes it difficult to build a generic transform to a relational database - such a transform would need to inspect the entire data set to see what the largest depth is. It also makes it impossible to allow editing, at least without placing some hard limit on the recursion. It is admittedly true that any strategy based on XQuery will also be limited to static data sets for the present, but update mechanisms for XQuery are already beginning to appear and are likely to become part of some future standard.

5 Implementation Strategy

In our investigation, we compare two possible implementation strategies to NXT Search, our existing implementation.

5.1 Using NXT’s stand-off format

The first strategy is to use XQuery directly on NXT’s stand-off data storage format. The bulk of the work here is in writing libraries of XQuery functions that correctly interpret NXT’s stand-off child links in order to allow navigation over the same primary axes as are used in XPath, but with multiple parenthood, and operating over NXT’s multiple files. The libraries can resolve the `XLinks` NXT uses both forwards and backwards. Backwards resolution requires functions that access the corpus metadata to find out which files could contain annotations that could stand in the correct relationship to the starting node. Built on top of this infrastructure would be functions which implement the NQL operators.

Resolving ancestors is a rather expensive operation which involves searching an entire coding file for links to a node with a specified identity. Additionally, if a query includes variables which are not bound to a particular type, this precludes the possibility of reducing the search space to particular coding files.

A drawback to using XPath to query a hierarchy which is serialised to multiple annotation files, is that much of the efficiency of XPath expressions can be lost through the necessity of resolving XLinks at every child or parent step of the expression. This means that even the descendant and ancestor axes of XPath may not be used directly but must be broken down into their constituent single-step axes.

In addition to providing a transparent interface for navigating the data, it may be necessary to provide additional indexing of the data, to increase efficiency and avoid the duplication of calculations. An alternative is to overcome the standoff nature of the data by resolving links explicitly, as described in the following section.

5.2 Using a redundant data representation

The second strategy makes use of the classic trade-off between memory and speed by employing a redundant data representation that is both easy to calculate from NXT's data storage format and ensures that most of the navigation required exercises common parts of XPath, since these are the operations upon which XQuery implementations will have concentrated their resources.

The particular redundancy we have in mind relies on NXT's concept of "knitting" data. In NXT's data model, every node may have multiple parents, but only one set of children. Where multiple parents exist, at most one will be in the same file as the child node, with the rest connected by `XLinks`. "Knitting" is the process of starting with one XML file and recursively following children and child links, storing the expanded result as an XML file. The redundant representation we used is then the smallest set of expanded files that contains within it every child link from the original data as an XML child. .

Although this approach has the advantage of using XPath more heavily than our first approach, it has the added costs of generating the knitted data and handling the redundancy. The knitting stylesheet that currently ships with NXT is very slow, but a very fast implementation of the knitting process that works with NXT format data has been developed and is expected as part of an upcoming LTXML release (University of Edinburgh Language Technology Group, nd). The cost of dealing with redundancy depends on the branching structure of the corpus. To date, most corpora with multiple parenthood have a number of quite shallow trees that do not branch themselves but all point to the same few base levels (e.g. orthography), suggesting we can at least avoid exponential expansion.

6 Tests

For initial testing, we chose a small set of queries which would allow us to judge potential implementations in terms of whether they could do everything we need to do, whether they would give the correct results, and how they would perform against our stated requirements. This allows us to form an opinion whilst only writing portions of the code required for a complete NQL implementation. Our set of queries is therefore designed to involve all of the basic operations required to exploit NXT's ability to represent multi-rooted trees and to traverse a large amount of data, so that they are computationally expensive and could return many results. In the tests, we ran the queries over the NXT translation for the Penn Treebank syntax annotated version of one Switchboard dialogue (Carletta et al., 2004), sw4114. The full dialogue is approximately 426Kb in physical size, and contains over 1101 word elements.

6.1 Test queries

Our test queries were as follows.

- Query 1 (Dominance):
  ```
  (exists $e nt)($w word):
  $e@cat="NP" && $e^$w
  ```
  (words dominated by an NP-category nt)

- Query 2 (Complex query with precedence and dominance):
  ```
  ($w1 word)($w2 word):
  TEXT($w1)="the" && $w1<>$w2
  ::
  (exists $p nt):  $p@cat="NP"
  && $p^$w1 && $p^$w2
  ```
  (word pairs where the word "the" precedes the second word with respect to a common NP dominator)

- Query 3 (Eliminative):
  ```
  ($a word)(forall $b
  turn):!($b^$a)
  ```
  (words not dominated by any turn)

In the data, the category "nt" represents syntactic non-terminals. The third query was chosen because it is particularly slow in the current NQL implementation, but is easily expressed as a path and therefore is likely to execute efficiently in XPath implementations.

Although NXT's object model also allows for arbitrary relationships between nodes using pointers with named roles, increasing speed for queries over them is only a secondary concern, and we know that implementing operators over them is possible in XQuery because it is very similar to resolving stand-off child links. For this reason, none of our test queries involve pointers.

6.2 Test environment

For processing XQuery, we used Saxon (www.saxonica.com), which provides an API so that it can be called from Java. There are several available XQuery interpreters, and they will differ in their implementation details. We chose Saxon because it appears to be most complete and is well-supported. Alternative interpreters, Galax (www.galaxquery.org) and Qexo (www.gnu.org/software/qexo/), provided only incomplete implementations at the time of writing.

6.3 Comparability of results

It is not possible in a test like this to produce completely comparable results because the different implementation strategies are doing very different things to arrive at their results. For example, consider our second query. Apart from some primitive optimizations, on this and all queries, NXT Search does an exhaustive search of all possible k-tuples that match the types given in the query, varying the rightmost variable fastest. Our XQuery implementation on stand-off data first finds matches to $w1, $w2, and $np; then calls a function that calculates the ancestries for matches to $w1 and $w2; for each ($w1, $w2) pair, computes the intersection of the two ancestries; and finally filters this intersection against the list of $np matches.

On the other hand, the implementation on the knitted data is shown in figure 1. It first sets variables representing the XML document containing our knitted data and all distinct nt elements within that document which both have a category attribute "NP" and have further `word` descendants. It then sets a variable to represent the sequence of results. The results are calculated by taking each `NP`-type element and checking its `word` descendants for those pairs where a word "the" precedes another word. The implementation also applies the condition that the `NP`-type element must not have another `NP` element as an ancestor — this is to remove duplicates introduced by the way we find the initial set of `NP`s.

In addition to the execution strategies, the methods used to start off processing were quite different. For each of the implementations, we did whatever gave the best performance. For the XQuery-based implementations, this meant writing a Java class to start up a static context for the execution of the query and reusing it to run the query repeatedly. For NXT, it meant using a shell script to run the command-line utility `SaveQueryResults` repeatedly on a set of observations, exiting each time.

Figure 1: An XQuery rewritten for knitted data; containing more direct XPath expressions.

```
let $doc := doc("data/knitted/swbd/sw4114.syntax.xml"),
    $nps := $doc//nt[@cat="NP"][descendant::word] union (),
    $result := (
        for $np in $nps return (
            let $w2 := $np//word, $w1 := $w2[text()="the"]
            for $a in $w1, $b in $w2
            where (struct:node-precedes($a, $b)
                and not($np/ancestor::nt[@cat="NP"]))
            (: only return for the uppermost common NP ancestor :)
            return (element match {$a, $b})
        )
    ) union ()
return element result {attribute count count($result), $result}
```

Our aim in performing the comparison is to assess what is *possible* in each approach rather than to do the same thing in each, and this is why we have attempted to achieve best possible performance in each context rather than making the conditions as similar as possible. In all cases, the figures we report are the mean timings over five runs of what the Linux `time` command reports as 'real' time.

7 Speed Results

The results of our trial are shown in the following table. Timings which are purely in seconds are given to 2 decimal places; those which extend into the minutes are given to the nearest second. "NXT" means NXT Search; "XQ" is the condition with XQuery using stand-off data; and "XQ-K" is the condition with XQuery using the redundant knitted data.

	Query		
Impl	Q1	Q2	Q3
NXT	3.38s	1m24	18.25s
XQ	10.21s	3m24	14.42s
XQ-K	2.03s	2.17s	2.47s

Although it would be wrong to read too much into our simple testing, these results do suggest some tentative conclusions. The first is that using XQuery on NXT's stand-off data format is unlikely to increase execution speed except for queries that are computationally very expensive for NXT, and may decrease performance for other queries. If users show any tolerance for delays, it is more likely to be for the delays to the former, and therefore this does not seem a winning strategy. On the other hand, using XQuery on the knitted data provides useful (and sometimes impressive) gains across the board.

It should be noted that our results are based upon a single XQuery implementation and are inevitably implementation-specific. Future work will also attempt to make comparisons with alternatives, including those provided by XML databases.

7.1 Memory results

To explore our second requirement, the ability to load more data, we generated a series of corpora which double in size from an initial set of 4 children with 2 parents.

We ran both NXT Search and XQuery in Saxon on these corpora, with the Java Virtual Machine initialised with increasing amounts of memory, and recorded the maximum corpus size each was able to handle. Both query languages were exercised on NXT stand-off data, with the simple task of calculating parent/child relationships. Results are shown in the following table.

	Max corpus size (nodes, disk space)	
Mem Mb	**NXT**	**XQuery/Saxon**
500	$3 * 2^{17}$, 28Mb	$3 * 2^{19}$, 111Mb
800	$3 * 2^{18}$, 56Mb	$3 * 2^{20}$, 224Mb
1000	$3 * 2^{18}$, 56Mb	$3 * 2^{20}$, 224Mb

These initial tests suggest that at its best, the XQuery implementation in Saxon can manage around 4 times as much data as NXT Search. It is interesting to note that the full set of tests took about 19 minutes for XQuery, but 18 hours for NXT Search. That is, Saxon appears to be far more efficient at managing large data sets. We also discovered that the NXT results were different when a different query was used; we hope to elaborate these results more accurately in the future.

We did not specifically run this test on the implementation that uses XQuery on knitted data because the basic characteristics would be the same as for the XQuery implementation with stand-off data. The size of a knitted data version will depend on the amount of redundancy that knitting creates. Knitting has the potential to increase the amount of memory required greatly, but it is worth noting that it does not always do so. The knitted version of the Switchboard dialogue used for these tests is actually smaller than the stand-off version, because the original stand-off stores terminals (words) in a separate file from syntax trees even though the terminals are defined to have only one parent. That is, there can be good reasons for using stand-off annotation, but it does have its own costs, as `XLinks` take space.

7.2 Query rewriting

In the testing described far, we used the existing version of NXT Search. Rather than writing a new query language implementation, we could just invest our resources in improvement of NXT Search itself. It is possible that we could change the underlying XML handling to use libraries that are more memory-efficient, but this is unlikely to give real scalability. The biggest speed improvements could probably be made by re-ordering terms before query execution. Experienced query authors can often speed up a query if they rewrite the terms to minimize the size of the search space, assuming they know the shape of the underlying data set. Although we do not yet have an algorithm for this rewriting, it roughly involves ignoring the “exists” quantifier, splitting the query into a complex one with one variable binding per subquery, sequencing the component queries by increasing order of match set size, and evaluating tests on the earliest subquery possible. For example, consider the query

```
  ($w1 word):text($w1)="the" ::
($p nt):$p@cat eq "NP" && $p^$w1 ::
($w2 word):  $p^$w2 && $w1<>$w2
```

This query, which bears a family resemblance to query 2, takes 4.31s, which is a considerable improvement. Of course, the result tree is a different shape from the one specified in the original query, and so this strategy for gaining speed improvements would incur the additional cost of rewriting the result tree after execution.

7.2.1 Discussion

Our testing suggests that if we want to make speed improvements, creating a new NQL implementation that uses XQuery on a redundant data representation is a good option. Although not the result we initially expected, it is perhaps unsurprising. This XQuery implementation strategy draws more heavily on XPath than the stand-off strategy, and XPath is the most well-exercised portion of XQuery. The advantages do not just come from recasting our computations as operations over trees. XPath allows us, for instance, to write a single expression that both binds a variable and performs condition tests on it, rather than requiring us to first bind the variable and then loop through each combination of nodes to determine which satisfy the constraints. Using a redundant data representation increases memory requirements, but the XQuery-based strategies use enough less memory that the redundancy in itself will perhaps not be an issue. In order to settle this question, we must think more carefully about the size and shape of current and potential NXT corpora.

Our other option for making speed improvements is to augment NXT Search with a query rewriting strategy. This needs further evaluation because the improvements will vary widely with the query being rewritten, but our initial test worked surprisingly well. However, augmenting the current NXT Search in this way will not reduce its memory use, and it is not clear whether this improvement can readily be made by other means.

Acknowledgments

This work has been funded by a grant from Scottish Enterprise via the Edinburgh-Stanford Link. We are grateful to Stefan Evert for designing NQL and for discussing its specification with us, and to Jan-Torsten Milde and Felix Sasaki for making available to us their own initial experiments suggesting that this re-implementation would be worth attempting.

References

[Bird et al.2006] Steven Bird, Yi Chen, Susan Davidson, Haejoong Lee, and Yifeng Zheng. 2006. Designing and evaluating an XPath dialect for linguistic queries. In *22nd International Conference on Data Engineering*, Atlanta, USA.

[Boag et al.2005] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, Jrme Simon, and Mugur Stefanescu. 2005. Xquery 1.0: An XML Query Language, November. http://www.w3.org/TR/xquery/; accessed 18 Jan 06.

[Carletta et al.2003] J. Carletta, Stefan Evert, Ulrich Heid, Jonathan Kilgour, Judy Robertson, and Holger Voormann. 2003. The NITE XML Toolkit: flexible annotation for multi-modal language data. *Behavior Research Methods, Instruments, and Computers*, 35(3):353–363.

[Carletta et al.2004] Jean Carletta, Shipra Dingare, Malvina Nissim, and Tatiana Nikitina. 2004. Using the NITE XML Toolkit on the Switchboard Corpus to study syntactic choice: a case study. In *Fourth Language Resources and Evaluation Conference*, Lisbon, Portugal.

[Carletta et al.in press] J. Carletta, S. Evert, U. Heid, and J. Kilgour. in press. The NITE XML Toolkit: data model and query language. *Language Resources and Evaluation Journal*.

[Clark and DeRose1999] James Clark and Steve DeRose. 1999. Xml path language (xpath) version 1.0, 16 November. http://www.w3.org/TR/xpath; accessed 18 Jan 06.

[Heid et al.2004] Ulrich Heid, Holger Voormann, Jan-Torsten Milde, Ulrike Gut, Katrin Erk, and Sebastian Pad. 2004. Querying both time-aligned and hierarchical corpora with NXT Search. In *Fourth Language Resources and Evaluation Conference*, Lisbon, Portugal.

[Iacob and Dekhtyar2005] Ionut E. Iacob and Alex Dekhtyar. 2005. Towards a query language for multihierarchical xml: Revisiting xpath. In *Eighth International Workshop on the Web and Databases (WebDB 2005)*, Baltimore, Maryland, USA, 16-17 June.

[National Institute of Standards and Technology2000] National Institute of Standards and Technology. 2000. ATLAS Project. http://www.nist.gov/speech/atlas/; last update 6 Feb 2003; accessed 18 Jan 06.

[Rohdend] Doug Rohde. n.d. Tgrep2. http://tedlab.mit.edu/ dr/Tgrep2/; accessed 18 Jan 06.

[Tiger Projectnd] Tiger Project. n.d. Linguistic interpretation of a German corpus. http://www.ims.uni-stuttgart.de/projekte/TIGER/; last update 17 Nov 2003; accessed 1 Mar 2004.

[University of Edinburgh Language Technology Groupnd] University of Edinburgh Language Technology Group. n.d. LTG Software. http://www.ltg.ed.ac.uk/software/; accessed 18 Jan 2006.