Constructing a Generic Natural Language Interface for an XML Database

Yunyao Li[1], Huahai Yang[2], H. V. Jagadish[1]

[1] University of Michigan, Ann Arbor, MI 48109, USA [*]
(yunyaol, jag)@eecs.umich.edu
[2] University at Albany, SUNY, Albany, NY 12222, USA
hyang@albany.edu

Abstract. We describe the construction of a generic natural language query interface to an XML database. Our interface can accept an arbitrary English sentence as a query, which can be quite complex and include aggregation, nesting, and value joins, among other things. This query is translated, potentially after reformulation, into an XQuery expression. The translation is based on mapping grammatical proximity of natural language parsed tokens in the parse tree of the query sentence to proximity of corresponding elements in the XML data to be retrieved. Our experimental assessment, through a user study, demonstrates that this type of natural language interface is good enough to be usable now, with no restrictions on the application domain.

1 Introduction

In the real world we obtain information by asking questions in a natural language, such as English. Supporting arbitrary natural language queries is regarded by many as the ultimate goal for a database query interface, and there have been numerous attempts towards this goal. However, two major obstacles lie in the way of reaching the ultimate goal of support for arbitrary natural language queries: first, automatically understanding natural language is itself still an open research problem, not just semantically but even syntactically; second, even if we could fully understand any arbitrary natural language query, translating this parsed natural language query into a correct formal query remains an issue since this translation requires mapping the understanding of intent into a specific database schema.

In this paper, we propose a framework for building a generic interactive natural language interface to database systems. Our focus is on the second challenge: given a parsed natural language query, how to translate it into a correct structured query against the database. The issues we deal with include those of attribute name confusion (e.g. asked "Who is the president of YMCA," we do not know whether YMCA is a country, a corporation, or a club) and of query structure confusion (e.g. the query "Return the lowest price for each book" is totally different from the query "Return the book with the lowest price," even though the words used in the two are almost the same). We address these issues in this paper through the introduction of the notions of *token attachment* and *token relationship* in natural language parse trees. We also propose the concept of *core token* as an effective mechanism to perform semantic grouping and hence determine both query nesting and structural relationships between result elements when mapping tokens to queries. Details of these notions can be found in Sec. 3.

[*] Supported in part by NSF 0219513 and 0438909, and NIH 1-U54-DA021519-01A1

Of course, the first challenge of understanding arbitrary natural language cannot be ignored. But a novel solution to this problem per se is out of the scope of this paper. Instead, we leverage existing natural language processing techniques, and use an off-the-shelf natural language parser in our system. We then extract semantics expressible by XQuery from the output of the parser, and whenever needed, interactively guide the user to pose queries that our system can understand by providing meaningful feedback and helpful rephrasing suggestions. Sec. 4 discusses how the system interacts with a user and facilitates query formulation during the query translation process.

We have incorporated our ideas into a working software system called NaLIX[3], which we evaluated by means of a user study. Our experimental results in Sec. 5 demonstrate the feasibility of such an interactive natural language interface to database systems. In most cases no more than two iterations appears to suffice for the user to submit a natural language query that the system can parse. Previous studies [4, 25] show that even casual users frequently revise queries to meet their information needs. Therefore, our system can be considered to be usable in practice. In NaLIX, a correctly parsed query is almost always translated into a structured query that correctly retrieves the desired answer (average precision = 95.1%, average recall = 97.6%).

Finally, we discuss related work in Sec. 6 and conclude in Sec. 7. We begin with some necessary background material in Sec. 2.

In summary, we have been able to produce a natural language query interface for a database that, while far from being able to pass the Turing test, is perfectly usable in practice, and able to handle even quite complex queries, e.g. involving nesting and aggregation, in a variety of application domains.

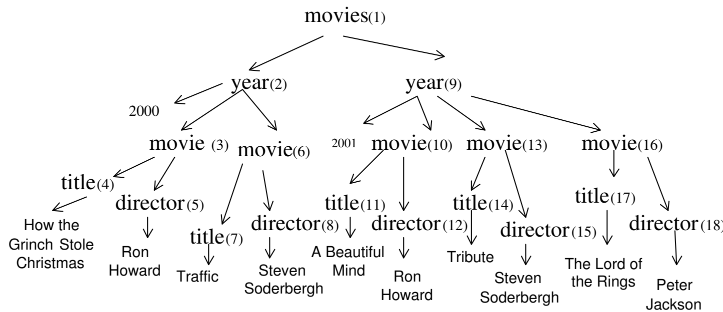
2 Background

Keyword search interfaces to databases have begun to receive increasing attention [6, 10–12, 16, 18], and can be considered a first step towards addressing the challenge of natural language querying. Our work builds upon this stream of research, so we present some essential background material here. Additional efforts at constructing natural language interfaces are described in Sec. 6.

There are two main ideas in using keyword search for databases. First, sets of keywords expressed together in a query must match objects that are "close together" in the database (using some appropriate notions of "close together"). Second, there is a recognition that pure keyword queries are rather blunt – too many things of interest are hard to specify. So somewhat richer query mechanisms are folded in along with the basic keyword search. A recent effort in this stream of work is Schema-Free XQuery [16, 18].

The central idea in Schema-Free XQuery is that of a *meaningful query focus* (MQF) of a set of nodes. Beginning with a given collection of keywords, each of which identifies a candidate XML element to relate to, the MQF of these elements, if one exists, automatically finds relationships between these elements, if any, including additional related elements as appropriate. For example, for the query "Find the director of Gone with the Wind," there may be *title* of *movie*, and *title* of *book* with value "Gone with the Wind" in the database. However, we do not need advanced semantic reasoning capability to know that only movies can have a director and hence "Gone with the Wind" should be the *title* of a *movie* instead of a *book*. Rather, the computation of mqf(*director*,

[3] NaLIX was demonstrated at SIGMOD 2005, and voted the Best Demo [17].



Query 1: *Return every director who has directed as many movies as has Ron Howard.*
Query 2: *Return every director, where the number of movies directed by the director is the same as the number of movies directed by Ron Howard.*
Query 3: *Return the directors of movies, where the title of each movie is the same as the title of a book.*

Fig. 1. Querying XML database with natural language queries

title) will automatically choose only *title* of *movie*, as this *title* has a structurally meaningful relationship with *director*. Furthermore, it does not matter whether the schema has *director* under *movie* or vice versa (for example, movies could have been classified based on their directors). In either case, the correct structural relationships will be found, with the correct *director* elements be returned.

Schema-Free XQuery greatly eases our burden in translating natural language queries in that it is no longer necessary to map the query to the precise underlying schema. We will use it as the target language of our translation process. From now on, we will refer to Schema-Free XQuery as XQuery for simplicity, unless noted otherwise.

3 From Natural Language Query To XQuery

The relationships between words in the natural language query must decide how the corresponding components in XQuery will be related to each other and thus the semantic meaning of the resulting query. We obtain such relationship information between parsed tokens from a dependency parser, which is based on the relationship between words rather than hierarchical constituents (group of words) [20, 28]. The parser currently used in NaLIX is Minipar [19]. The reason we chose Minipar is two-fold: (i) it is a state-of-art dependency parser; (ii) it is free off-the-shelf software, and thus allows easier replication of our system.

There are three main steps in translating queries from natural language queries into corresponding XQuery expressions. Sec. 3.1 presents the method to identify and classify terms in a parse tree output of a natural language parser. This parse tree is then validated, but we defer the discussion of this second step until Sec. 4. Sec. 3.2 demonstrates how a validated parse tree is translated into an XQuery expression. These three key steps are independent of one another; improvements can be made to any one without impacting the other two. The software architecture of NaLIX has been described in [17], but not the query transformation algorithms. Figure 1 is used as our running example to illustrate the query transformation process.

Table 1. Different Types of Tokens

Type of Token	**Query Component**	**Description**
Command Token(CMT)	Return Clause	Top main verb or wh-phrase [24] of parse tree, from an enum set of words and phrases
Order by Token(OBT)	Order By Clause	A phrase from an enum set of phrases
Function token(FT)	Function	A word or phrase from an enum set of adjectives and noun phrases
Operator Token(OT)	Operator	A phrase from an enum set of preposition phrases
Value Token(VT)	Value	A noun or noun phrase in quotation marks, a proper noun or noun phrase, or a number
Name token(NT)	Basic Variable	A non-VT noun or noun phrase
Negation (NEG)	function not()	Adjective "not"
Quantifier Token(QT)	Quantifier	A word from an enum set of adjectives serving as determiners

Table 2. Different Types of Markers

Type of Marker	**Semantic Contribution**	**Description**
Connection Marker(CM)	Connect two related tokens	A preposition from an enumerated set, or non-token main verb
Modifier Marker(MM)	Distinguish two NTs	An adjective as determiner or a numeral as predeterminer or postdeterminer
Pronoun Marker(PM)	None due to parser's limitation	Pronouns
General Marker(GM)	None	Auxiliary verbs, articles

3.1 Token Classification

To translate a natural language query into an XQuery expression, we first need to identify words/phrases in the original sentence that can be mapped into corresponding components of XQuery. We call each such word/phrase a *token*, and one that does not match any component of XQuery a *marker*. Tokens can be further divided into different types shown in Table 1 according to the type of query components they match.[4] Enumerated sets of phrases (enum sets) are the real-world "knowledge base" for the system. In NaLIX, we have kept these small - each set has about a dozen elements. Markers can be divided into different types depending on their semantic contribution to the translation. A unique id is assigned to each token or marker. The parse tree after token identification for Query 2 in Figure 1 is shown in Figure 2. Note that node **11** is not in the query, nor in the output of the parser. Rather, it is an *implicit* node (formally defined in Sec. 4) that has been inserted by the token validation process.

Note that because of the vocabulary restriction of the system, some terms in a query may not be classified into one of the categories of token or marker. Obviously, such unclassified terms cannot be properly mapped into XQuery. Sec. 4 describes how these are reported to the user during parse tree validation, when the relationship of the "unknown" terms with other tokens (markers) can be better identified.

3.2 Translation into XQuery

Given a valid parse tree (discussion on parse tree validation is deferred until Sec. 4), we show here how to translate it into XQuery. XML documents are designed with the goal to be "human-legible and reasonably clear." [32] Therefore, any reasonably designed XML document should reflect certain semantic structure isomorphous to human conceptual structure, and hence expressible by human natural language. The challenge is to utilize the structure of the natural language constructions, as reflected in the parse tree, to generate appropriate structure in the XQuery expression (If we do not establish this structure, then we may as well just issue a simple keyword query!!). For simplicity of presentation, we use the symbol for each type of token (resp. marker) to refer to tokens

[4] When a noun/noun phrase matches certain XQuery keywords, such as "string", special handling is required. Such special cases are not listed in the table, and will not be discussed in the paper due to space limitation.

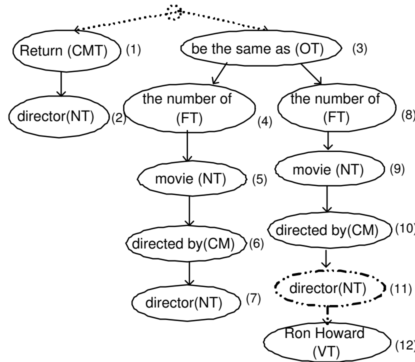


Fig. 2. Parse tree for Query 2 in Figure 1

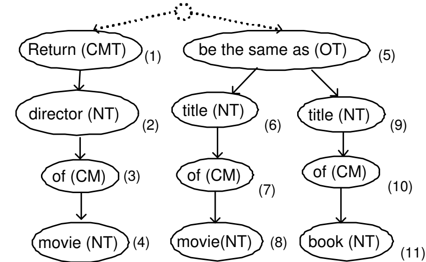


Fig. 3. Parse tree for Query 3 in Figure 1

(markers) of that type, and use subscripts to distinguish different tokens (markers) of the same type if needed. For instance, we will write, "Given NT_1, NT_2, ..." as a short hand for "Given name tokens u and v, ..."

3.2.1 Concepts and Definitions.

A natural language query may contain multiple name tokens (NTs), each corresponding to an element or attribute in the database. NTs "related" to each other should be mapped into the same **mqf** function in Schema-Free XQuery and hence found in structurally related elements in the database. However, this relationship among the NTs is not straightforward. Consider the example in Figure 3, nodes **2** (*director*) and **4** (*movie*) should be considered as related to nodes **6** (*title*) and **8** (*movie*), since the two *movie* nodes (**4**, **8**) are semantically equivalent. However, they are not related to nodes **9** (*title*) or **11** (*book*), although the structural relationship between nodes **9**, **11** and nodes **2**, **4** is exactly the same as that between nodes **6**, **8** and nodes **2**, **4**. An intuitive explanation for this distinction is that the two sets of NTs (*director*, *movie*) and (*title*, *movie*) are related to each other semantically because they share NTs representing the same *movie* elements in the database, whereas the (*title*, *book*) pair does not. We now capture this intuition formally.

Definition 1 (Name Token Equivalence). *NT_1 and NT_2 are said to be equivalent if they are (i) both not implicit* [5] *and composed of the same noun phrase with equivalent modifiers*[6]*; OR (ii) both implicit and correspond to VTs of the same value.*

In consequence of the above definition, if a query has two occurrences of *book*, the corresponding name tokens will be considered equivalent, if they are not qualified in any way. However, we distinguish *first book* from *second book*: even though both correspond to *book* nodes, the corresponding name tokens are not equivalent, since they have different modifiers.

Definition 2 (Sub-Parse Tree). *A subtree rooted at an operator token node that has at least two children is called a sub-parse tree.*

[5] An implicit NT is a NT not explicitly included in the query. It is formally defined in Definition 11, Sec. 4.

[6] Two modifiers are obviously equivalent if they are the same. But some pairs of distinct modifiers may also be equivalent. We do not discuss modifier equivalence further in this paper for lack of space.

Definition 3 (Core Token). *A name token is called a core token if (i) it occurs in a sub-parse tree and has no descendant name tokens;* OR *(ii) it is equivalent to a core token.*

Definition 4 (Directly Related Name Tokens). *NT_1 and NT_2 are said to be directly related to each other, if and only if they have a parent-child relationship (ignoring any intervening markers, and FT and OT nodes with a single child).*

Definition 5 (Related by Core Token). *NT_1 and NT_2 are said to be related by core token, if and only if they are directly related to the same or equivalent core tokens.*

Definition 6 (Related Name Tokens). *NT_1 and NT_2 are said to be related, if they are directly related, or related by core token, or related to the same NT.*

For Query 3 in Figure 3, only one operator token (OT), node **5** exists in the parse tree. The lowest NTs of the OT's sub-parse trees, nodes **8** (*movie*) and **11** (*book*), are the core tokens in the query. Nodes **2**, **6** and **9** are directly related to nodes **4**, **8** and **11** respectively, by Definition 4. Node **4** is equivalent to node **8**. Hence, according to Definition 6, two sets of related nodes {**2, 4, 6, 8**} and {**9, 11**} can be obtained.

All NTs related to each other should be mapped to the same **mqf** function since we seek elements (and attributes) matching these NTs in the database that are structurally related.

Additional relationships between tokens (not just name tokens) needed for query translation are captured by the following definition of *attachment*.

Definition 7 (Attachment). *Given any two tokens T_a and T_b, where T_a is the parent of T_b in the parse tree (ignoring all intervening markers), if T_b follows T_a in the original sentence, then T_a is said to attach to T_b; otherwise, T_b is said to attach to T_b.*

3.2.2 Token Translation.

Given the conceptual framework established above, we describe in this section how each token in the parse tree is mapped into an XQuery fragment. The mapping process has several steps. We illustrate each step with our running example.

Identify Core Token Core tokens in the parse tree are identified according to Definition 3. Two different core tokens can be found in Query 2 in Figure 1. One is *director*, represented by nodes **2** and **7**. The other is a different *director*, represented by node **11**. Note although node **11** and nodes **2**, **7** are composed of the same word, they are regarded as different core tokens, as node **11** is an implicit NT, while nodes **2**, **7** are not.

Variable Binding Each NT in the parse tree should be bound to a basic variable in Schema-Free XQuery. We denote such variable binding as: $\langle var \rangle \rightarrow NT$

Two name tokens should be bound to different basic variables, unless they are regarded as the same core token, or identical by the following definition:

Definition 8 (Identical Name Tokens). *NT_1 and NT_2 are identical, if and only if (i) they are equivalent, and indirectly related;* AND *(ii) the NTs directly related with them, if any are identical;* AND *(iii) no FT or QT attaching to either of them.*

We then define the relationships between two basic variables based on the relationships of their corresponding NTs as follows:

Table 3. Variable Bindings for Query 2

Variable	Associated Content	Nodes	Related To
$\$v_1^*$	*director*	**2,7**	$\$v_2$
$\$v_2$	*movie*	**5**	$\$v_1$
$\$v_3$	*movie*	**9**	$\$v_4$
$\$v_4^*$	*director*	**11**	$\$v_3$
$\$cv_1$	count($\$v_2$)	**4+5**	N/A
$\$cv_2$	count($\$v_4$)	**8+9**	N/A

Table 4. Direct Mapping for Query 2

Pattern	Query Fragment
$\$v_1$	for $\$v_1$ in $\langle doc\rangle$//director
$\$v_2$	for $\$v_2$ in $\langle doc\rangle$//movie
$\$v_3$	for $\$v_3$ in $\langle doc\rangle$//movie
$\$v_4$	for $\$v_4$ in $\langle doc\rangle$//director
$\$cv_1+\langle eq\rangle+\$cv_2$	where $\$cv_1 = \$cv_2$
$\$v_4+\langle constant\rangle$	where $\$v_4$ = "Ron Howard"
$\langle return\rangle + \$v_1$	return $\$v_1$

Definition 9 (Directly Related Variables). *Two basic variables $\langle var_1\rangle$ and $\langle var_2\rangle$ are said to be directly related, if and only if for any NT_1 corresponding to $\langle var_1\rangle$, there exists a NT_2 corresponding to $\langle var_2\rangle$ such that NT_1 and NT_2 are directly related, and vice versa.*

Definition 10 (Related Variables). *Two basic variables $\langle var_1\rangle$ and $\langle var_2\rangle$ are said to be related, if and only if any NTs corresponding to them are related or there is no core token in the query parse tree*

Patterns $\langle FT + NT\rangle|\langle FT_1 + FT_2 + NT\rangle$ should also be bound to variables. Variables bound with such patterns are called *composed variable*s, denoted as $\langle cmpvar\rangle$, to distinguish them from the basic variables bound to NTs. We denote such variable binding as:

$$\langle function\rangle \rightarrow FT$$
$$\langle cmpvar\rangle \rightarrow (\langle function\rangle + \langle var\rangle)|(\langle function\rangle + \langle cmpvar\rangle)$$

Table 3 shows the variable bindings[7] for Query 2 in Figure 1. The nodes referred to in the table are from the parse tree of Query 2 in Figure 2.

Mapping Certain patterns of tokens can be mapped directly into clauses in XQuery. A complete list of patterns and their corresponding clauses in XQuery can be found in Figure 4. Table 4 shows a list of direct mappings from token patterns to query fragments for Query 2 in Figure 1 ($\rightsquigarrow$ is used to abbreviate 'translates into').

3.2.3 Grouping and Nesting.

The grouping and nesting of the XQuery fragments obtained in the mapping process has to be considered when there are function tokens in the natural language query, which correspond to aggregate functions in XQuery, or when there are quantifier tokens, which correspond to quantifiers in XQuery. Determining grouping and nesting for aggregate functions is difficult, because the scope of the aggregate function is not always obvious from the token it directly attaches to. Determining grouping and nesting for quantifiers is comparatively easier.

Consider the following two queries: "Return the lowest price for each book," and "Return each book with the lowest price." For the first query, the scope of function min() corresponding to "lowest" is within each book, but for the second query, the scope of function min() corresponding to "lowest" is among all the books. We observe that *price*, the NT the aggregate function attaching to, is related to *book* in different ways in the two queries. We also notice that the CM "with" in the second query implies that a *price* node related to *book* has the same value as the lowest price of all the *book*s. Based on the above observation, we propose the transformation rules shown in Figure 5 to take the semantic contribution of connection markers into consideration.

[7] The * mark next to $\$v_1$, $\$v_4$ indicates that the corresponding NTs are core tokens.

- **FOR clause:**
  Let basic() be the function that returns the name token corresponding to basic variable in $\langle var \rangle$ or $\langle cmpvar \rangle$
  $\langle var \rangle \rightsquigarrow$ <u>for</u> $\langle var \rangle$ <u>in</u> $\langle doc \rangle$//basic($\langle var \rangle$)
- **WHERE clause:**
  $\langle variable \rangle \rightarrow \langle var \rangle | \langle cmpvar \rangle$
  $\langle constant \rangle \rightarrow VT$
  $\langle arg \rangle \rightarrow \langle variable \rangle | \langle constant \rangle$
  $\langle opr \rangle \rightarrow OT$
  $\langle neg \rangle \rightarrow NEG$
  $\langle quantifier \rangle \rightarrow QT$
  $\langle var \rangle + \langle constant \rangle \rightsquigarrow$ <u>where</u> $\langle var \rangle = \langle constant \rangle$
  $(\langle variable \rangle + \langle opr \rangle + \langle arg \rangle) | (\langle opr \rangle + \langle var \rangle + \langle constant \rangle) \rightsquigarrow$ <u>where</u> $\langle variable \rangle + \langle opr \rangle + \langle arg \rangle$
  $\langle variable \rangle + \langle neg \rangle + \langle opr \rangle + \langle arg \rangle \rightsquigarrow$ <u>where</u> <u>not</u> $(\langle variable \rangle + \langle opr \rangle + \langle arg \rangle)$
  $\langle opr \rangle + \langle constant \rangle + \langle variable \rangle \rightsquigarrow \langle cmpvar \rangle \rightarrow$ <u>count</u>($\langle variable \rangle$)
  <u>where</u> $\langle cmpvar \rangle + \langle opr \rangle + \langle constant \rangle$
  $\langle neg \rangle + \langle opr \rangle + \langle constant \rangle + \langle variable \rangle \rightsquigarrow \langle cmpvar \rangle \rightarrow$ <u>count</u>($\langle variable \rangle$)
  <u>where</u> <u>not</u> $(\langle cmpvar \rangle + \langle opr \rangle + \langle constant \rangle)$
- **ORDERBY clause:**
  $\langle sort \rangle \rightarrow OBT$
  $\langle sort \rangle + \langle variable \rangle \rightsquigarrow$ <u>orderby</u> $\langle variable \rangle$
- **RETURN clause:**
  $\langle cmd \rangle \rightarrow CMT$
  $\langle cmd \rangle + \langle variable \rangle \rightsquigarrow$ <u>return</u> $\langle variable \rangle$

Fig. 4. Mapping from token patterns to query fragments

Let innerFT() be function returning innermost FT in $\langle cmpvar \rangle$

$\langle connector \rangle \rightarrow CM$
$\langle cmpvar \rangle \rightarrow FT + \langle var_2 \rangle$
$\langle var_1 \rangle + \langle connector \rangle + \langle cmpvar \rangle \rightsquigarrow \quad \langle var_2 \rangle_{new} \rightarrow$ basic($\langle cmpvar \rangle$)
if innerFT($\langle cmpvar \rangle$) $\neq$ **null**, **then**
<u>where</u> $\langle$innerFT($\langle cmpvar \rangle$)$\rangle + \langle var_2 \rangle_{new} = \langle cmpvar \rangle$
else
<u>where</u> $\langle var_2 \rangle_{new} = \langle cmpvar \rangle$

Record $\langle var_2 \rangle_{new}$ as related to $\langle var_1 \rangle$, $\langle var_2 \rangle$ as unrelated to $\langle var_1 \rangle$

Fig. 5. Semantic contribution of connection marker in query translation

We then propose the mapping rules shown in Figure 6 to determine the nesting scope for aggregate functions. Specifically, we identify two different nesting scopes that result from using an aggregate function - *inner* and *outer*, with respect to the basic variable $\langle var \rangle$ that the function directly attaches to. The nesting scope of the LET clause corresponding to an aggregate function depends on the basic variable that it attaches to. The idea is that if an aggregate function attaches to a basic variable that represents a core token, then all the clauses containing variables related to the core token should be put inside the LET clause of this function; otherwise, the relationships between name tokens (represented by variables) via the core token will be lost. For example, given the query "Return the total number of movies, where the director of each movie is Ron Howard," the only core token is *movie*. Clearly, the condition clause "<u>where</u> \$dir = 'Ron Howard'" should be bound with each *movie* inside the LET clause. Therefore, the nesting scope of a LET clause corresponding to the core token is marked as *inner* with respect to $\langle var \rangle$ (in this case \$movie). On the other hand, if an aggregate function attaches to a basic variable $\langle var \rangle$ representing non-core token, only clauses containing variables directly related to $\langle var \rangle$ should be put inside of the LET clause, since $\langle var \rangle$ is only associated with other variables related to it via a core token. The nesting scope of the LET clause should be marked as *outer*, with respect to $\langle var \rangle$. Similarly, when there is no core token, $\langle var \rangle$ may only be associated with other variables indirectly related to it via value joins. The nesting scope of the LET clause should also be marked as *outer* with respect to $\langle var \rangle$. In such a case, the nesting scope determination for Query 2 can

```
Denote ⟨core⟩ as the core token related to ⟨var⟩, if any; else as a variable ⟨var⟩
attaching to and directly related to, if any; else as a randomly chosen variable
indirectly related to ⟨var⟩.
Denote ⟨v⟩ as variables directly related to ⟨var⟩.

if ⟨cmpvar⟩ → ⟨function⟩+⟨var⟩
   then ⟨cmpvar⟩ ⇝
  – if ⟨var⟩ is not a core token itself, or there is no core token, then
              let ⟨vars⟩ := {
                for  ⟨core1⟩ in  ⟨doc⟩//basic(⟨core⟩)
                where   ⟨core1⟩ = ⟨core⟩
                return   ⟨var⟩}
    Replace ⟨cmpvar⟩ with ⟨function⟩ + ⟨vars⟩.
    Mark ⟨var⟩ and ⟨core⟩, ⟨v⟩ and ⟨core⟩ as unrelated.
    Mark ⟨var⟩ and ⟨core1⟩, ⟨v⟩ and ⟨core1⟩ as related.
    Mark nesting scope for the LET clause as outer with respect to ⟨var⟩.

  – else if ⟨var⟩ is a core token itself, or no ⟨core⟩ exists, then
             let ⟨vars⟩ := { return   ⟨var⟩}
    Replace ⟨cmpvar⟩ with ⟨function⟩ + ⟨vars⟩.
    Mark nesting scope for the LET clause as inner with respect to ⟨var⟩.
else if ⟨cmpvar⟩ → ⟨function⟩+⟨cmpvar1⟩
   then ⟨function⟩+⟨cmpvar1⟩ ⇝
           let ⟨vars⟩ := {⟨cmpvar1⟩}
   Recursively rewrite ⟨cmpvar1⟩.
   Replace ⟨cmpvar⟩ with ⟨function⟩ + ⟨vars⟩.
```

Fig. 6. Grouping and nesting scope determination for aggregate functions

be found in Figure 8. The updated variable bindings and relationships between basic variables for the query can be found in Table 5.

The nesting scope determination for a quantifier (Figure 7) is similar to that for an aggregate function, except that the nesting scope is now associated with a quantifier inside a WHERE clause. The nesting scope of a quantifier is marked as *inner* with respect to $\langle var \rangle$ the quantifier attaching to, when the variable $\langle var \rangle$ is a core token. Otherwise, it is marked as *outer* with respect to $\langle var \rangle$. The meanings of *inner* and *outer* are the same as those for aggregate functions, except that now only WHERE clauses may be put inside of a quantifier.

MQF Function As we have previously discussed in Sec. 3.2, all name tokens related to each other should be mapped into the same **mqf** function. Hence, basic variables corresponding to such name tokens should be put into the same **mqf** function. One WHERE clause containing **mqf** function can be obtained for each set of related basic variables:

$\langle vars \rangle \rightarrow$ the union of all $\langle var \rangle s$ related to each other
$\langle vars \rangle \rightsquigarrow$ where mqf($\langle vars \rangle$)

As can be seen from Table 5, two sets of related variables can be found for Query 2 in Figure 1: $\{\$v_5,\$v_2\}$ and $\{\$v_3,\$v_6\}$. The corresponding WHERE clauses containing **mqf** function are: where mqf($\$v_5,\$v_2$) and where mqf($\$v_3,\$v_6$).

3.2.4 Full Query Construction.

Multiple XQuery fragments may be obtained from token translation. These fragments alone do not constitute a meaningful query. We need to construct a semantically meaningful Schema-Free XQuery by putting these fragments together with appropriate nestings and groupings.

```
/*⟨core⟩ is the same as that defined in Figure 6*/
if ⟨cmpvar⟩ → ⟨quantifier⟩+⟨var⟩
   then ⟨cmpvar⟩ ⇝
  – if ⟨var⟩ is not a core token itself, or there is no core token, then
            let ⟨vars⟩ := {
              for ⟨core1⟩ in ⟨doc⟩//basic(⟨core⟩)
              where ⟨core1⟩ = ⟨core⟩
              return ⟨var⟩}
            where ⟨quantifier⟩ ⟨var1⟩ in ⟨vars⟩ satisfies { }
    Mark ⟨var⟩ and ⟨core,⟩, ⟨core1⟩ as unrelated.
    Replace ⟨var⟩ elsewhere with ⟨var1⟩, except in FOR clause.
    Mark nesting scope for the WHERE clause with the quantifier as outer with
    respect to ⟨var⟩.
  – else if ⟨var⟩ is a core token itself, or no ⟨core⟩ exists, then
           let ⟨vars⟩ := { return ⟨var⟩}
           where ⟨quantifier⟩ ⟨var1⟩ in ⟨vars⟩ satisfies { }
    Mark nesting scope for the WHERE clause with the quantifier as inner with
    respect to ⟨var⟩.
    Replace ⟨var⟩ elsewhere with ⟨var1⟩, except in FOR clause.
```

Fig. 7. Grouping and nesting scope determination for quantifier

```
(1) $cv1→count($v2)
   $v2 is not a core token, and the core
token related to it is $v1, therefore
$cv1 ⇝
       let $vars1 := {
         for $v5 in ⟨doc⟩//director
         where  mqf($v2,$v5)
            and $v5 = $v1
         return $v2}
   Replace all $cv1 with count($vars1).
   Mark $v2, $v1 as unrelated.
   Mark $v2, $v5 as related.
   Mark nesting scope for the LET
clause as outer with respect to $v2.
```

```
(2) $cv2→count($v3)
   $v3 is not a core token, and the core
token related to it is $v4, therefore
$cv1 ⇝
       let $vars2 := {
         for $v6 in ⟨doc⟩//director
         where  mqf($v3,$v6)
            and $v6 = $v4
         return $v3}
   Replace all $cv2 with count($vars2).
   Mark $v3, $v4 as unrelated.
   Mark $v3, $v6 as related.
   Mark nesting scope for the LET
clause as outer with respect to $v3.
```

Fig. 8. Grouping and nesting scope determination in Query 2

Following the defined nesting scopes (Figure 6,7), we construct the query starting from innermost clauses and work outwards. If the scope defined is *inner* with respect to $\langle var \rangle$, then all the other query fragments containing $\langle var \rangle$ or basic variables related to $\langle var \rangle$ are put within an inner query following the FLOWR convention (e.g., conditions in WHERE clauses are connected by and) as part of the query at outer level. If the scope defined is *outer* with respect to $\langle var \rangle$, then only queries fragments containing $\langle var \rangle$, and clauses (in case of quantifier, only WHERE clauses) containing basic variables directly related to $\langle var \rangle$ are put inside the inner query, while query fragments of other basic variables indirectly related to $\langle var \rangle$ are put outside of the clause at the same level of nesting. The remaining clauses are put in the appropriate places at the outmost level of the query following the FLOWR convention. Full translation for Query 2 in Figure 1 can be found in Figure 9.

4 Interactive Query Formulation

The mapping process from natural language to XQuery requires our system to be able to map words to query components based on token classification. Due to the limited vo-

```
for $v1 in doc("movie.xml")//director,
    $v4 in doc("movie.xml")//director
let $vars1 := {
    for $v5 in doc("movie.xml")//director,
        $v2 in doc("movie.xml")//movie
    where  mqf($v2,$v5)
       and $v5 = $v1
    return  $v2}
let $vars2 := {
    for $v6 in doc("movie.xml")//director,
        $v3 in doc("movie.xml")//movie
    where  mqf($v3,$v6)
       and $v6 = $v4
    return  $v3}
where  count($vars1) = count($vars2)
   and $v4 = "Ron Howard"
return $v1
```

Fig. 9. Full translation for Query 2

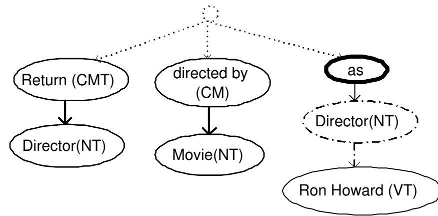


Fig. 10. Parse tree for Query 1 in Figure 1

Table 5. Updated variable bindings for Query 2

Variable	Associated Content	Nodes	Related To
$\$v_1^*$	*director*	**2,7**	**null**
$\$v_2$	*movie*	**5**	$\$v_5$
$\$v_3$	*movie*	**9**	$\$v_6$
$\$v_4^*$	*director*	**11**	**null**
$\$v_5^*$	*director*	N/A	$\$v_2$
$\$v_6^*$	*director*	N/A	$\$v_3$
$\$cv_1$	count($vars$_1$)	**4**+**5**	N/A
$\$cv_2$	count($vars$_2$)	**8**+**9**	N/A

Table 6. Grammar supported by NaLIX
Symbol "+" represents attachment relation between two tokens; "[]" indicates implicit token, as defined in Def. 11

1. Q → RETURN PREDICATE* ORDER_BY?
2. RETURN → CMT+(RNP|GVT|PREDICATE)
3. PREDICATE → QT?+((RNP$_1$|GVT$_1$)+GOT+(RNP$_2$|GVT$_2$)
4. |(GOT?+RNP+GVT)
5. |(GOT?+GVT+RNP)
6. |(GOT?+[NT]+GVT)
7. |RNP
8. ORDER_BY → OBT+RNP
9. RNP → NT |(QT+RNP)|(FT+RNP)|(RNP∧RNP)
10. GOT → OT|(NEG+OT)|(GOT∧GOT)
11. GVT → VT |(GVT∧GVT)
12. CM → (CM+CM)

cabulary understood by the system, certain terms cannot be properly classified. Clever natural language understanding systems attempt to apply reasoning to interpret these terms, with partial success. We make no attempt at superior understanding of natural language. Rather, our approach is to get the user to rephrase the query into terms that we can understand. By doing so, we shift some burden of semantic disambiguation from the system to the user, to whom such task is usually trivial. In return, the user obtains better accessibility to information via precise querying.

To ensure that this process proceeds smoothly for the user, we provide the user with specific feedback on how to rephrase. In this section we describe the validation process we use to determine whether we can translate a user specified query. We also discuss the informative error messages we produce when validation fails.

NaLIX is designed to be a query interface for XML by translating natural language queries into Schema-Free XQuery. As such, the linguistic capability of our system is essentially restricted by the expressiveness of XQuery. This is to say, a natural language query that may be understood and thus meaningfully mapped into XQuery by NaLIX is one whose semantics is expressible in XQuery. Furthermore, for the purpose of evaluating the query, only the semantics that can be expressed by XQuery need to be extracted and mapped into XQuery.

Consider the following query: "Find all the movies directed by director Ron Howard." The meaning of "directed by" cannot be directly expressed in XQuery. It is neither possible nor necessary for NaLIX to understand such semantics. Instead based on the dependency parse tree, we can determine that "movie" and "director" are related and should be mapped into the same **mqf** function. Then the structural relationship between *movie* and *director* nodes in the database, which corresponds to "directed by," will be properly captured by Schema-Free XQuery. Generally, the semantics extracted

by NaLIX from a given natural langauge query comprise two parts: (i) tokens that can be directly mapped into XQuery; (ii) semantic relationships between tokens, which are inexpressible in XQuery, but are reflected by database schema, such as the attachment relation between "movie" and "director" via "directed by" in the above example.

The grammar for natural language corresponding to the XQuery grammar supported by NaLIX is shown in Table 6 (ignoring all markers).We call a normalized parse tree that satisfies the above grammar a *valid* parse tree.

A valid parse tree can be translated to an XQuery expression as described in Sec 3.2. An invalid parse tree, however, will be rejected by the system, with error message(s).[8] Each error message is dynamically generated, tailored to the actual query causing the error. Inside each message, possible ways to revise the query are also suggested. For example, Query 1 in Figure 1 is found to be an invalid query, since it contains an unknown term "as" as highlighted in the parse tree in Figure 10. An error message will be returned to the user, and suggest "the same as" as a possible replacement for "as." Query 3 in Figure 3 is likely to be the new query written by the user by using the suggested term "the same as." Screenshots of the above iteration can be found in [17]. By providing such meaningful feedback tailored to each particular query instance, we eliminate the need to require users to study and remember tedious instructions on the system's linguistic coverage. Instead, through such interactive query formulation process, a user will gradually learn the linguistic coverage of the system. Note that we assume user queries are written in correct English, and thus do not specify any rules to deal with incorrect English.

For some queries, the system successfully parses and translates the queries, yet may not be able to correctly interpret the user's intent. These queries will be accepted by the system, but with warnings. For example, determining pronoun references (the "anaphora" resolution problem) remains an issue in natural language processing. Whenever there exists a pronoun in a user query, we include a warning message in the feedback and let the user be aware of the potential misunderstanding.

During the validation process, we also perform the following additional tasks concerned with database specific situations.

Term Expansion A user may not be familiar with the specific attributes and element names in the database. Therefore, a name token specified in the user query may be different from the actual name(s) of element or attribute in the database matching this particular name token. The task of finding the name(s) of element or attribute in the database that matches with a given name token is accomplished by ontology-based term expansion using generic thesaurus WordNet [36] and domain-specific ontology whenever one is available.

Implicit Name Token In a natural language query, we may find value tokens where the name tokens attaching to them are implicit in the query. For example, in Query 1 of Figure 10, element *director* in the database is related to value token "Ron Howard," but is not explicitly included in the query. We call such name tokens *implicit name token* as defined below. See Table 6 for the definitions of GVT, GOT and RNP.

Definition 11 (Implicit Name Token). *For any GVT, if it is not attached by a CMT, nor adjacent to a RNP, nor attached by a GOT that is attached by a RNP or GVT, then each VT within the GVT is said to be related to an implicit NT (denoted as [NT]). An implicit NT related to a VT is the name(s) of element or attribute with the value of VT in the database.*

[8] More details on the generation of error and warning messages in NaLIX can be found on the Web at http://www.umich.edu/ yunyaol/NaLIX/index.html.

If no name matching a name token or the value of a value token can be found in the database, an error message will be returned. If multiple element or attribute with different names matching the name token or value token are found in the database, the disjunctive form of the names is regarded as the corresponding name for the given name token, or implicit name token for the given value token. Users may also change the query by choosing one or more of the actual names.

5 Experimental Evaluation

We implemented NaLIX as a stand-alone interface to the Timber native XML database [13, 33] that supports Schema-Free XQuery. We used Minipar [19] as our natural language parser. To evaluate the relative strength of NaLIX, we experimentally compared it with a keyword search interface that supports search over XML documents based on Meet [26]. We would have liked to compare NaLIX with an existing NLP system. Unfortunately, existing NLP systems are mainly designed for textual content, not for structured data. As such, NLP question answering system cannot handle queries as complex as NaLIX and we believe no meaningful comparison is possible.

5.1 Methods

Participants were recruited with flyers posted on a university campus. Eighteen of them completed the full experiment. Their age ranged from 19 to 55 with an average of 27. A questionnaire indicated that all participants were familiar with some form of keyword search (e.g. Google) but had little knowledge of any formal query language.

Procedures. The experiment was a within-subject design, with each participant using either NaLIX or keyword search interface in one experimental block. The order of the two blocks was randomly assigned for each participant. Within each block, each participant was asked to accomplish 9 search tasks in a random order determined by a pair of orthogonal 9 by 9 Latin Squares.

The search tasks were adapted from the "XMP" set in the XQuery Use Cases [31]. Each search task was described with the elaborated form of an "XMP" query[9] taken from XQuery Use Cases [31]. Participants received no training at all on how to formulate a query, except being instructed to use either an English sentence or some keywords as the query depending on which experiment block the participant was in.

We noted that in an experimental setting, a participant could be easily satisfied with poor search quality and go on to the next search task. In order to obtain objective measurement of interactive query performance, a search quality criteria was adopted. Specifically, the results of a participant's query were compared against a standard results set, upon which precision and recall were automatically calculated. A harmonic mean of precision and recall [27] greater than 0.5 was set as passing criteria, beyond which the participant may move on to the next task. To alleviate participants' frustration and fatigue from repeated passing failures, a time limit of 5 minutes was set for each task. If a participant reached the criteria before the time limit, he or she was given the choice to move on or to revise the query to get better results.

[9] Q12 is not included, as set comparison is not yet supported in Timber. Q5 is not included, as NaLIX current only supports queries over a single document. Q11 contains two separate search tasks: the second task was used as Q11 in our experiment; the first task, along with Q2, is the same as Q3, and thus is not included, as they only differ in the form of result display, which is not the focus of NaLIX.

Measurement. We evaluated our system on two objective metrics: how hard it was for the users to specify a query (*ease of use*); and how good was the query produced in terms of retrieving correct results (*search quality*).

Ease of Use For each search task, we recorded the number of iterations and the actual time (from the moment the participant started a search task by clicking on a button) it took for a participant to formulate a system-acceptable query that returned the best results (i.e., highest harmonic mean of precision and recall) within the time limit for the task. We also evaluated NaLIX subjectively by asking each participant to fill out a post-experiment questionnaire.

Search Quality The quality of a query was measured in terms of accuracy and comprehensiveness using standard precision and recall metrics. The correct results for each search task is easy to obtain given the corresponding correct schema-aware XQuery. Since the expected results were sometimes complex, with multiple elements (attributes) of interest, we considered each element and attribute value as an independent value for the purposes of precision and recall computation. Thus, a query that returned all the right elements, but only 3 out of 4 attributes requested for each element, would have a recall score of 75%. Ordering of results was not considered when computing precision and recall, unless the task specifically asked the results be sorted.

Finally, we measured the time NaLIX took for query translation and the time Timber took for query evaluation for each query. Both numbers were consistently very small (less than one second), and so not of sufficient interest to be worth reporting here. The fast query translation is expected, given that query sentences were themselves not very large. The fast evaluation time is an artifact of the miniscule data set that was used. The data set we used was a sub-collection of DBLP, which included all the elements on books in DBLP and twice as many elements on articles. The total size of the data set is 1.44MB, with 73142 nodes when loaded into Timber. We chose DBLP because it is semantically close to the data set for the XMP use case such that the “XMP” queries can be applied with only minor changes (e.g., tag name *year* is used to replace *price*, which is not in the data set but has similar characteristics). A pilot study showed that slow system response times (likely with very large data sets) resulted in frustration and fatigue for the participants. Since query evaluation time is not a focus of this paper, we felt that it is most appropriate to use this data set to balance the trade-off between performance and realism: we minimized the overhead resulting from the use of a larger data set both in terms of query evaluation and precision/recall computation time; at the same time, the correct results obtained for any “XMP” query from our data set were the same as those would have been obtained by using the whole DBLP, as correct answers for each query included elements related to *book* elements only.

5.2 Results and Discussion

Ease of Use. The time and the number of iterations needed for participants to formulate a valid natural language query with the best search results is shown in Figure 11. As can be seen, the average total time needed for each search task is usually less than 90 seconds, including the time used to read, understand the task description, mentally formulate a query, type in the query, read the feedback message, revise the query, browse the results and decide to accept the results. In consequence, there seems to be a floor of about 50 seconds, which is the average minimum time required for any query. The average number of iterations needed for formulating a query acceptable by NaLIX is less than 2, with an average of 3.8 iterations needed for the worst query. For about half of

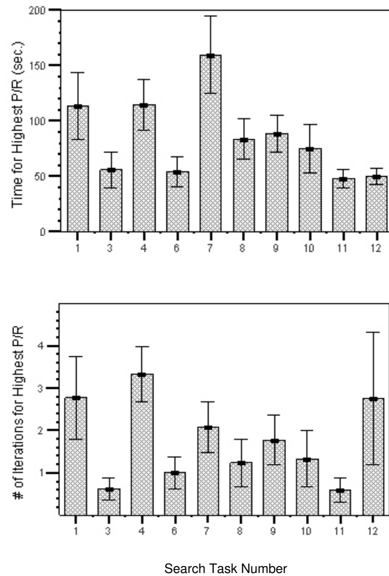


Fig. 11. Average time (in *sec.*) and average number of iterations needed for each "XMP" search task. Error bars show standard errors of means

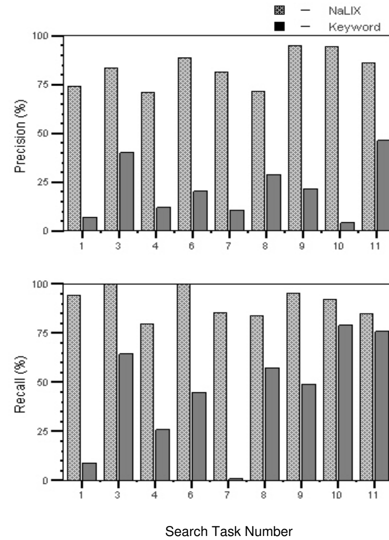


Fig. 12. Average precision and recall for each "XMP" search task

the search tasks (not the same tasks for each participant), all the participants were able to formulate a natural language query acceptable by NaLIX on the first attempt (i.e., with zero iterations). Also, for each task, there was at least one user (not the same one each time) who had an acceptable phrasing right off the bat (i.e. the minimum number of iterations was zero for each task).

It is worth noting that there was no instance where a participant became frustrated with the natural language interface and abandoned his/her query attempt. However, two participants decided to stop the experiment due to frustration during the keyword search block.

According to the questionnaire results, the users felt that simple keyword search would not have sufficed for the query tasks they had to do. They welcomed the idea of a natural language query interface, and found NaLIX easy to use. The average participants' levels of satisfaction with NaLIX was 4.11 on a scale of 1 to 5, where 5 denotes "extremely easy to use."

Search quality. Figure 12 compares the average precision and recall of NaLIX with that of a keyword search interface in the experiment. As can be seen, the search quality of natural language queries was consistently better than that of keyword search queries. The precision of NaLIX is 83.0% on average, with an average precision of 70.9% for the worst query; for 2 out of the 9 search tasks, NaLIX achieved perfect recall, with an average recall of 90.1% for all the queries and an average recall of 79.4% for the worst query. In contrast, keyword search performed poorly on most of the search tasks[10],

[10] Each search task corresponds to an "XMP" query in [31] with the same task number.

Table 7. Average Precision and Recall

	avg.precision	avg.recall	total queries
all queries	83.0%	90.1%	162
all queries specified correctly	91.4%	97.8%	120
all queries specified parsed correctly	95.1%	97.6%	112

especially on those requiring complex manipulations such as aggregation or sorting (e.g. Q7, Q10). Even for queries with simple constant search conditions and requiring no further manipulation (e.g. Q4, Q11), keyword searches produced results that were less than desirable.

In our experiments, we found two major factors contributing to search quality loss for NaLIX. First, the participants sometimes failed to write a natural language query that matched the exact task description. For instance, one of the users expressed Q6 as "List books with title and authors" (rather than only list the title and authors of the book), resulting in a loss of precision. The second had to do with parsing error. Given a generic natural language query, it is sometimes difficult to determine what exactly should be returned, and the parse tree obtained may be incorrect.[11] For example, one of the users formulated Q1 as "List books published by Addison-Wesley after 1991, including their year and title." Minipar wrongly determined that only "book" and "title" depended on "List," and failed to recognize the conjunctive relationship between "year" and "title." Consequently, NaLIX failed to return *year* elements in the result, resulting in a loss of both precision and recall. Table 7 presents summary statistics to tease out the contributions of these two factors. If one considers only the 112 of 162 queries that were specified and parsed correctly, then the error rate (how much less than perfect are the precision and recall) is roughly reduced by 75%, and NaLIX achieved average precision and recall of 95.1% and 97.6%, respectively, in the experiments.

6 Related Work

In the information retrieval field, research efforts have long been made on natural language interfaces that take keyword search query as the target language [5, 8]. In recent years, keyword search interfaces to databases have begun to receive increasing attention [6, 10–12, 16, 18], and have been considered a first step towards addressing the challenge of natural language querying. Our work builds upon this stream of research. However, our system is not a simple imitation of those in information retrieval field in that it supports a richer query mechanism that allow us to convey much more complex semantic meaning than pure keyword search.

Extensive research has been done on developing natural language interfaces to databases (NLIDB), especially during the 1980's [2]. The architecture of our system bears most similarity to syntax-based NLIDBs, where the resulting parse tree of a user query is directly mapped into a database query expression. However, previous syntax-based NLIDBs, such as LUNAR [35], interface to application-specific database systems, and depend on the database query languages specially designed to facilitate the mapping from the parse tree to the database query [2]. Our system, in contrast, uses a generic query language, XQuery, as our target language. In addition, unlike previous systems such as the one reported in [29], our system does not rely on extensive domain-specific knowledge.

[11] Minipar achieves about 88% precision and 80% recall with respect to dependency relations with the SUSANNE Corpus [19].

The idea of interactive NLIDB has been discussed in some early NLIDB literature [2, 15]. The majority of these focus on generating cooperative responses using query results obtained from a database with respect to a user's task(s). In contrast, the focus of the interactive process of our system is purely query formulation: only one query is actually evaluated against the database. There has also been work to build interactive query interfaces to facilitate query formulation [14, 34]. These works depend on domain-specific knowledge. Also, they assist the construction of structured queries rather than natural language queries.

There are a few notable recent works on NLIDB ([21–23, 30]). A learning approach as a combination of learning methods is presented in [30]. We view such learning approaches and our approach as complimentary to each other - while learning techniques may help NaLIX to expand its linguistic coverage, NaLIX can provide training sources for a learning system. A NLIDB based on a query formulator is described in [21]. A statistical approach is applied to determine the meaning of a keyword. The keywords can then be categorized into query topics, selection list, and query constraints as the input of query formulator. No experimental evaluation on the effectiveness of the system has been reported. PRECISION [22, 23] is a NLIDB that translates *semantically tractable* NL questions into corresponding SQL queries. While PRECISION extensively depends on database schema for query mapping, NaLIX does not rely on the availability of a schema for query translation. In addition, PRECISION requires each database attribute be manually assigned with a compatible *wh-value*, while NaLIX does not. Finally, NaLIX covers a much broader range of natural language questions than PRECISION with promising quality.

In NaLIX, we obtain the semantic relationships between words via a dependency parser. Recent work in question answering [3, 7, 9] has pointed out the value of utilizing the dependency relation between words in English sentence to improve the precision of question answering. Such dependency relations are obtained either from dependency parsers such as Minipar [3, 7] or through statistic training [9]. These works all focus on full text retrieval, and thus cannot directly apply to XML databases. Nevertheless, they inspired us to use a dependency parser to obtain semantic relationship between words, as we have done in NaLIX.

7 Conclusion and Future Work

We have described a natural language query interface for a database. A large class of natural language queries can be translated into XQuery expressions that can then be evaluated against an XML database. Where natural language queries outside this class are posed, an interactive feedback mechanism is described to lead the user to pose an acceptable query. The ideas described in this paper have been implemented, and actual user experience gathered. Our system as it stands supports comparison predicates, conjunctions, simple negation, quantification, nesting, aggregation, value joins, and sorting. In the future, we plan to add support for disjunction, for multi-sentence queries, for complex negation, and for composite result construction. Our current system is oriented at structured XML databases: we intend to incorporate support for phrase matching by incorporating full-text techniques in XQuery such as TeXQuery [1], thereby extending our applicability to databases primarily comprising text stored as XML.

The system as we have it, even without all these planned extensions, is already very useful in practice. We already have a request for production deployment by a group outside computer science. We expect the work described in this paper to lead to a whole new generation of query interfaces for databases.

References

1. S. Amer-Yahia et al. TeXQuery: A full-text search extension to XQuery. In *WWW*, 2004.
2. I. Androutsopoulos et al. Natural language interfaces to databases - an introduction. *Journal of Language Engineering*, 1(1):29–81, 1995.
3. G. Attardi et al. PiQASso: Pisa question answering system. In *TREC*, 2001.
4. M. J. Bates. The design of browsing and berrypicking techniques for the on-line search interface. *Online Review*, 13(5):407–431, 1989.
5. J. Chu-carroll et al. A hybrid approach to natural language Web search. In *EMNLP*, 2002.
6. S. Cohen et al. XSEarch: A semantic search engine for XML. In *VLDB*, 2003.
7. H. Cui et al. Question answering passage retrieval using dependency relations. In *SIGIR*, 2005.
8. S. V. Delden and F. Gomez. Retrieving NASA problem reports: a case study in natural language information retrieval. *Data & Knowledge Engineering*, 48(2):231–246, 2004.
9. J. Gao et al. Dependency language model for information retrieval. In *SIGIR*, 2004.
10. L. Guo et al. XRANK: Ranked keyword search over XML documents. In *SIGMOD*, 2003.
11. V. Hristidis et al. Keyword proximity search on XML graphs. In *ICDE*, 2003.
12. A. Hulgeri et al. Keyword search in databases. *IEEE Data Engineering Bulletin*, 24:22–32, 2001.
13. H. V. Jagadish et al. Timber: A native xml database. *The VLDB Journa*, 11(4):274–291, 2002.
14. E. Kapetanios and P. Groenewoud. Query construction through meaningful suggestions of terms. In *FQAS*, 2002.
15. D. Kupper et al. NAUDA: A cooperative natural language interface to relational databases. *SIGMOD Record*, 22(2):529–533, 1993.
16. Y. Li et al. Schema-Free XQuery. In *VLDB*, 2004.
17. Y. Li et al. NaLIX: an interactive natural language interface for querying XML. In *SIGMOD*, 2005.
18. Y. Li et al. Enabling Schema-Free XQuery with Meaningful Query Focus. *To appear in VLDB Journal*, 2006.
19. D. Lin. Dependency-based evaluation of MINIPAR. In *Workshop on the Evaluation of Parsing Systems*, 1998.
20. I. A. Mel'čuk. *Studies in dependency syntax*. Karoma Publishers, Ann Arbor, MI, 1979.
21. F. Meng and W. Chu. Database query formation from natural language using semantic modeling and statistical keyword meaning disambiguation. Technical Report 16, UCLA, 1999.
22. A.-M. Popescu et al. Towards a theory of natural language interfaces to databases. In *IUI*, 2003.
23. A.-M. Popescu et al. Modern natural language interfaces to databases: Composing statistical parsing with semantic tractability. In *COLING*, 2004.
24. R. Quirk et al. *A Comprehensive Grammar of the English Language*. Longman, London, 1985.
25. J. R. Remde et al. Superbook: an automatic tool for information exploration - hypertext? In *Hypertext*, pages 175–188. ACM Press, 1987.
26. A. Schmidt et al. Querying XML documents made easy: Nearest concept queries. *ICDE*, 2001.
27. W. Shaw Jr. et al. Performance standards and evaluations in IR test collections: Cluster-based retrieval modles. *Information Processing and Management*, 33(1):1–14, 1997.
28. D. Sleator and D. Temperley. Parsing English with a link grammar. In *International Workshop on Parsing Technologies*, 1993.
29. D. Stallard. A terminological transformation for natural language question-answering systems. In *ANLP*, 1986.
30. L. R. Tang and R. J. Mooney. Using multiple clause constructors in inductive logic programming for semantic parsing. In *ECML*, 2001.
31. The World Wide Web Consortium. XML Query Use Cases. W3C Working Draft. Available at http://www.w3.org/TR/xquery-use-cases, 2003.
32. The World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (Third Edition). W3C Recommendation. Available at http://www.w3.org/TR/REC-xml, 2004.
33. Timber: http://www.eecs.umich.edu/db/timber.
34. A. Trigoni. Interactive query formulation in semistructured databases. In *FQAS*, 2002.
35. W. Woods et al. *The Lunar Sciences Natural Language Information System: Final Report*. Bolt Beranek and Newman Inc., Cambridge, MA, 1972.
36. WordNet: http://www.cogsci.princeton.edu/∼wn.