NaLIX: A Generic Natural Language Search Environment for XML Data

YUNYAO LI
IBM Almaden Research Center
HUAHAI YANG
University at Albany, State University of New York
and
H. V. JAGADISH
University of Michigan

We describe the construction of a generic natural language query interface to an XML database. Our interface can accept a large class of English sentences 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. Iterative search in the form of followup queries is also supported. 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.

Categories and Subject Descriptors: H.2.3 [**Database Management**]: Languages—*Query languages*; H.4.0 [**Information Systems Applications**]: General

General Terms: System, Design, Algorithms, Experimentation

Additional Key Words and Phrases: Natural language interface, iterative search, XQuery, XML, dialog system

This work was supported in part by NSF grant IIS 0438909 and NIH grants R01 LM008106 and U54 DA021519.
An earlier version of this article was presented at the 2006 International Conference on Extending Database Technology (EDBT'06, March 26–31, Munich, Germany).
Work done while Y. Li was a student at the University of Michigan.
Authors' addresses: Y. Li (contact author), IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120; email: yunyaoli@us.ibm.com; H. Yang, University at Albany, State University of New York, Draper Hall, Room 113, Western Avenue, Albany, NY 12222; email: hyang@albany.edu; H. V. Jagadish, Computer Science and Engineering Building, 2260 Hayward Street, Ann Arbor, MI 48109; email: jag@umich.edu.

ACM Reference Format:
Li, Y., Yang, H., and Jagadish, H. V. NaLIX: A generic natural language search environment for XML data. ACM Trans. Datab. Syst. 32, 4, Article 30 (November 2007), 44 pages. DOI = 10.1145/1292609.1292620 http://doi.acm.org/10.1145/1292609.1292620

1. INTRODUCTION

In the real world we obtain information by asking questions in a natural language, such as English. Not surprisingly, supporting arbitrary natural language queries is regarded by many as the ultimate goal for a database query interface, and there have been numerous attempts toward 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 would remain an issue since this translation requires mapping the understanding of intent into a specific database schema.

In this article, 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 article 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 Section 3.

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 article. 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. Section 4 discusses how the system interacts with a user and facilitates query formulation during the query translation process.

Search is rarely a one-step process: a user often needs to iteratively modify prior queries to obtain desired results. To facilitate such iterative search, we provide query history and query template to allow easy reuse of prior queries. More importantly, we also support followup queries that are partially specified as refinement to a prior query in the form of a separate query. Section 5 describes how followup queries are supported in the system.

We have incorporated our ideas into a working software system called NaLIX.[1] Figure 17 shows the screenshots of NaLIX in action. We evaluated the system by means of an experimental user study. Our experimental results in Section 6 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. This performance is acceptable in practice since previous studies [Bates 1989; Remde et al. 1987] have shown that even casual users frequently revise queries to meet their information needs. 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 Section 7 and conclude in Section 8. We begin with some necessary background material in Section 2.

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

2. BACKGROUND

Keyword search interfaces to databases have begun to receive increasing attention [Hulgeri et al. 2001; Cohen et al. 2003; Guo et al. 2003; Hristidis et al. 2003; Li et al. 2004, 2007c], and can be considered a first step toward 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 Section 7.

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 [Li et al. 2004, 2007c]. 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 $\underline{\text{mqf}}$(*director*, *title*) will automatically choose only *title* of *movie*, as this *title* has a structurally meaningful relationship with

[1]NaLIX was demonstrated at SIGMOD 2005, and voted the Best Demo [Li et al. 2005].

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

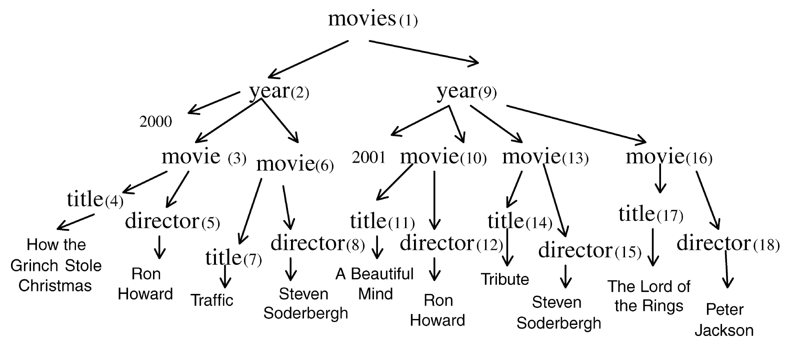
The relationships between words in the natural language query must decide how the corresponding components in XQuery will be related to each other and thus the semantic meaning of the resulting query. We obtain such relationship information between parsed tokens from a dependency parser, which is based on the relationship between words rather than hierarchical constituents (group of words) [Mel'čuk 1979; Sleator and Temperley 1993]. The parser currently used in NaLIX is MINIPAR [Lin 1998]. 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. Section 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 Section 4. Section 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. Figure 2 presents a bird's eye view of the translation process. Figure 1 is used as our running example to illustrate the query transformation process.

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 as shown in Table I according to the type of query components they match.[2] 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 3. Note that node 11

[2]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 article due to space limitations.



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.

```
1. Given a natural language query $S_Q$
2. Obtain parse tree $T_Q$ for $S_Q$ from a dependency parser
Step 1:
3. Classify words/phrases in $T_Q$ into tokens and markers based on Table I and II
Step 2:
4. Validate $T_Q$ based on the grammar supported by NaLIX (Table VI)
Step 3:
5. if $T_Q$ is valid then
6.      Assign variables for name tokens in $T_Q$
7.      Map parse tree fragments in $T_Q$ into XQuery fragments
8.      Determine groupings and nestings
9.      Construct full XQuery expression $Q$
10.       return $Q$ and MessageGenerator.Warnings($T_Q$)
11. else
12.     return MessageGenerator.Errors($T_Q$)
```

Fig. 2. Translation process overview: from natural language query to XQuery.

is not in the query, nor in the output of the parser. Rather, it is an *implicit* node (formally defined in Section 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. Section 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 (a discussion of parse tree validation is deferred until Section 4), we show here how to translate it into XQuery. XML documents are

Table I. Different Types of Tokens

Type of Token	Query Component	Description	Examples
Command Token (CMT)	Return Clause	Top main verb or wh-phrase [Quirk et al. 1985] of parse tree, from an enum set of words and phrases	*return*, *what is*
Order by Token (OBT)	Order By Clause	A phrase from an enum set of phrases	*sorted by*
Function Token (FT)	Function	A word or phrase from an enum set of adjectives and noun phrases	*maximum*
Operator Token (OT)	Operator	A phrase from an enum set of preposition phrases	*more than*
Value Token (VT)	Value	A noun or noun phrase in quotation marks, a proper noun or noun phrase, or a number	*Traffic*
Name Token (NT)	Basic Variable	A noun or noun phrase that is not a value token	*director*
Negation Token (NEG)	function not()	Adjective "not"	*not*
Quantifier Token (QT)	Quantifier	A word from an enum set of adjectives serving as determiners	*every*

Table II. Different Types of Markers

Type of Marker	Semantic Contribution	Description	Examples
Connection Marker (CM)	Connect two related tokens	A preposition from an enumerated set, or nontoken main verb	*of*, *directed*
Modifier Marker (MM)	Distinguish two name tokens	An adjectives as determiner or a numeral as predetermine or postdeterminer	*many*, *popular*
Pronoun Marker (PM)	None due to parser's limitation	Pronouns	*this*, *that*, *him*
General Marker (GM)	None	Auxiliary verbs, articles	*a*, *an*, *the*

designed with the goal of being "human-legible and reasonably clear" [World Wide Web Consortium 2004]. 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!!). In particular, we need to be able to take advantage of the semantic expressiveness of natural language and handle complex query semantics such as aggregation and quantifier, which can be easily specified in natural language, but cannot be easily expressed in other user-friendly alternatives for formal database query languages [Androutsopoulos et al. 1995]. For simplicity of presentation, we use the symbol for each type of token (respectively

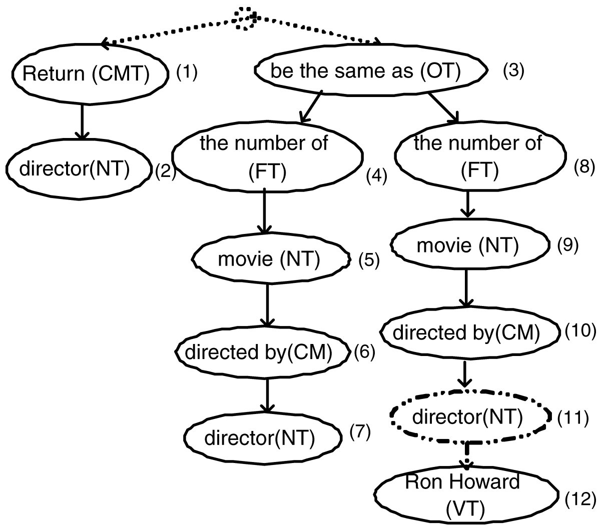


Fig. 3. Parse tree for Query 2 in Figure 1 (symbols in parenthesis are defined in Tables I and II).

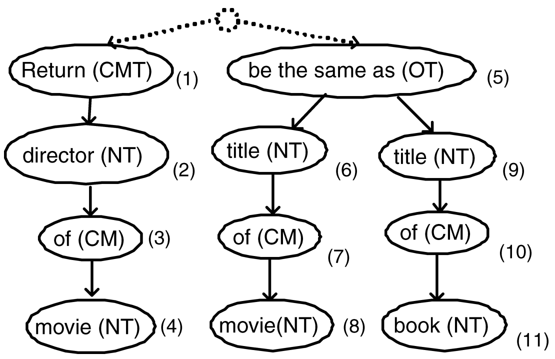


Fig. 4. Parse tree for Query 3 in Figure 1 (symbols in parenthesis are defined in Tables I and II).

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

3.2.1 *Concepts and Definitions.*

We first define the basic notions that we use to explore the existing structure of a natural language query for the purpose of query translation.

A natural language query may contain multiple name tokens, each corresponding to an element or an attribute in the database. For example, the query in Figure 4 contains several name tokens (labeled *NT*). Name tokens "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 name tokens is not straightforward. Consider the example in Figure 4: 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 name tokens (*director*, *movie*) and (*title*, *movie*) are related to each other semantically because they share name tokens representing the same *movie* elements in the database, whereas the (*title*, *book*) pair does not. We now capture this intuition formally.

Definition 3.1 (*Name Token Equivalence*). NT_1 and NT_2 are said to be *equivalent* if they are (i) both not implicit[3] and composed of the same noun phrase with equivalent modifiers,[4] or (ii) both implicit and corresponding to value tokens 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.

Elements/attributes specified in the same XQuery statement are related either via structural join or value join. Elements/attrbiutes related via structural join are structurally related to each other in the database, while elements/attributes that are related solely via value join typically are not structurally related. Only elements/attributes that are structurally related should be mapped into the same **mqf** function. As a result, in the translation process from natural language query to XQuery, special attention needs to be paid to operator tokens with two children name tokens. Such an operator token corresponds to value join and thus potentially introduces more than one group of related name tokens to the query. Below, we define the notion of *subparse tree* to capture the existence of value join.

When the same natural language query contains more than one group of related name tokens, each group of related name tokens typically has a name token that can be used to differentiate this group of name tokens from others. In a dependency parse tree, the lower a name token is in the parse tree, the further it constrains the query semantics, and thus the more important it is. As such, the lowest name tokens in a sub-parse tree are the name tokens to differentiate the different groups of name tokens, if any, in the parse tree. For instance, in the example in Figure 4, nodes 8 (*movie*) and 11 (*book*) helps us to identify the two groups of related nodes: (2, 4, 6, 8) and (9, 11). We capture such important name tokens by the notion of *core token*.

Definition 3.2 (*Core Token*). A *core token* is a name token that (i) occurs in a subparse tree and has no descendant name tokens, or (ii) is equivalent to a core token.

[3]An implicit name token is a name token not explicitly included in the query. It is formally defined in Definition 4.1 Section 4.

[4]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 article for lack of space.

We can further define different types of relationship between two name tokens based on how they are related to each other in the parse tree. The dependency relation between two name tokens indicates their semantic relatedness to each other. For instance, in the example in Figure 4, nodes having a dependency relation with each other—(2, 4), (6, 8), and (9, 11)—are regarded as related to each other, respectively. We refer to name tokens related to each other via dependency relations as *directly related*.

Definition 3.3 (*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).

In addition, in the above example, as we have discussed earlier, nodes 2, 4, 6, and 8 should be regarded as related as well, because the two *movie* nodes 4 and 8 are equivalent core tokens. In general, if two name tokens are related to the same or equivalent name token, then they are also regarded as related to each other, unless they are related to different core tokens. We can therefore define the following types of relationship between name tokens.

Definition 3.4 (*Related by Core Tokens*). 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 3.5 (*Related Name Tokens*). NT_1 and NT_2 are said to be *related*, if they are directly related to each other, related by core token, or related to the same name token, or there exists no core token.

As a result of Definition 3.5, if no core token exists in a natural language query, all the name tokens in the same query are regarded as related to each other. For Query 3 in Figure 4, only one operator token, node 5, exists in the parse tree. The lowest name tokens in the operator's subparse 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 3.3. Node 4 is equivalent to node 8. Hence, according to Definition 3.5, two sets of related nodes {2, 4, 6, 8} and {9, 11} can be obtained.

All name tokens related to each other should be mapped to the same **mqf** function since we seek elements (and attributes) matching these name tokens 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 3.6 (*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_a.

The notion of attachment allows us to distinguish children of the same node in the parse tree based on their original word order. Consider the example in Figure 4, node 5 (*be the same as*) is the parent of both nodes 6 (*title*) and 7 (*title*). However, based on on the above definition, node 5 attaches to node 6, while node 6 attaches to node 7.

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.

3.2.2.1 *Identify Core Token.* Core tokens in the parse tree are identified according to Definition 3.2. Two different core tokens can be found for Query 3 in Figure 1. One is *movie*, represented by nodes 4 and 8 (Figure 4). The other is *book*, represented by node 11. Two different core tokens can also be found for Query 2 in Figure 1. One is *director*, represented by nodes 2 and 7 (Figure 3). 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 name token, while nodes 2, 7 are not.

3.2.2.2 *Variable Binding.* Each name token in the parse tree should be bound to a basic variable in Schema-Free XQuery. We denote such variable binding as $NT \Rightarrow \langle var \rangle$, where $\Rightarrow$ stands for "map into."

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 3.7 (*Identical Name Tokens*). NT_1 and NT_2 are *identical*, if and only if (i) they are equivalent, related to each other but not directly related, (ii) the name token directly related with them, if any, are identical, and (iii) no function token or quantifier token attaches to either of them.

Note that, in the above definition, we distinguish two equivalent name tokens that are directly related from each other. The intuition is that directly related name tokens correspond to objects in the database that are structurally related. Since, in XML documents, the same object cannot be related to itself, the two directly related name tokens are not likely to refer to the same object in the database, and thus should not be regarded as identical.

We then define the relationships between two basic variables based on the relationships of their corresponding name tokens as follows.

Definition 3.8 (*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 3.9 (*Related Variables*). Two basic variables $\langle var_1 \rangle$ and $\langle var_2 \rangle$ are said to be *related*, if and only if any *NT*s corresponding to them are related or there is no core token in the query parse tree.

We will discuss later in this section how the above variable relationships are used to decide how query fragments containing different variables are put together into a full meaningful query.

Patterns $\langle FT + NT \rangle$, $\langle FT_1 + FT_2 + NT \rangle$, and $\langle QT + NT \rangle$ should also be bound to variables. Variables bound with such patterns are called *composed variables*, denoted as $\langle cmpvar \rangle$, to distinguish them from the basic variables bound to

Table III. 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

$NT \Rightarrow \langle var \rangle$
$FT \Rightarrow \langle function \rangle$
$QT \Rightarrow \langle quantifier \rangle$
$(\langle function \rangle + \langle var \rangle)|(\langle function_1 \rangle + \langle function_2 \rangle + \langle var \rangle) \Rightarrow \langle cmpvar \rangle$
$\langle quantifier \rangle + \langle var \rangle \Rightarrow \langle cmpvar \rangle$
$VT \Rightarrow \langle constant \rangle$
$OT \Rightarrow \langle opr \rangle$
$NEG \Rightarrow \langle neg \rangle$
$QT \Rightarrow \langle quantifier \rangle$
$OBT \Rightarrow \langle sort \rangle$
$CMT \Rightarrow \langle cmd \rangle$

Fig. 5. Mapping from tokens to XQuery components (symbols on the left hand side are defined in Table I).

name tokens. We denote such variable binding as

$FT \Rightarrow \langle function \rangle$
$QT \Rightarrow \langle quantifier \rangle$
$(\langle function \rangle + \langle var \rangle)|(\langle function_1 \rangle + \langle function_2 \rangle + \langle var \rangle) \Rightarrow \langle cmpvar \rangle$
$(\langle quantifier \rangle + \langle var \rangle) \Rightarrow \langle cmpvar \rangle$

Table III shows the variable bindings[5] for Query 2 in Figure 1. The nodes referred to in the table are from the parse tree of Query 2 in Figure 3.

3.2.2.3 *Mapping.* Similar to variable binding, certain tokens can be directly mapped into XQuery components. A complete list of such mappings is presented in Figure 5. Certain patterns of the XQuery components can then be directly mapped into clauses in XQuery. A complete list of patterns and their corresponding clauses in XQuery can be found in Figure 6. Table IV 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 must be considered when there are function tokens in the natural language query, which correspond to aggregation functions in XQuery, or when there are quantifier tokens, which correspond to quantifiers in XQuery. Determining grouping and nesting for aggregation functions is difficult, because the scope of an aggregation function is not

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

Let $\langle variable \rangle$ stand for $\langle var \rangle | \langle cmpvar \rangle$ and $\langle arg \rangle$ stand for $\langle variable \rangle | \langle constant \rangle$. Let NT be the name token corresponding to $\langle var \rangle$.

FOR clause:

$\langle var \rangle \rightsquigarrow \underline{\text{for}}\ \langle var \rangle\ \underline{\text{in}}\ \langle doc \rangle//NT$

WHERE clause:

$\langle var \rangle + \langle constant \rangle \rightsquigarrow \underline{\text{where}}\ \langle var \rangle = \langle constant \rangle$

$((\langle variable \rangle + \langle opr \rangle + \langle arg \rangle) | (\langle opr \rangle + \langle var \rangle + \langle constant \rangle)) \rightsquigarrow \underline{\text{where}}\ \langle variable \rangle + \langle opr \rangle + \langle arg \rangle$

$\langle variable \rangle + \langle neg \rangle + \langle opr \rangle + \langle arg \rangle \rightsquigarrow \underline{\text{where}}\ \underline{\text{not}}\ (\langle variable \rangle + \langle opr \rangle + \langle arg \rangle)$

$\langle opr \rangle + \langle constant \rangle + \langle variable \rangle \rightsquigarrow \langle cmpvar \rangle \rightarrow \underline{\text{count}}(\langle variable \rangle)$
$\qquad \underline{\text{where}}\ \langle cmpvar \rangle + \langle opr \rangle + \langle constant \rangle$

$\langle neg \rangle + \langle opr \rangle + \langle constant \rangle + \langle variable \rangle \rightsquigarrow \langle cmpvar \rangle \rightarrow \underline{\text{count}}(\langle variable \rangle)$
$\qquad \underline{\text{where}}\ \underline{\text{not}}\ (\langle cmpvar \rangle + \langle opr \rangle + \langle constant \rangle)$

ORDERBY clause:

$\langle sort \rangle + \langle variable \rangle \rightsquigarrow \underline{\text{orderby}}\ \langle variable \rangle$

RETURN clause:

$\langle cmd \rangle + \langle variable \rangle \rightsquigarrow \underline{\text{return}}\ \langle variable \rangle$

Fig. 6. Mapping from token patterns to XQuery fragments.

Table IV. Direct Mapping for Query 2

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

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 <u>min</u>() corresponding to "lowest" is within each book, but for the second query, the scope of function <u>min</u>() corresponding to "lowest" is among all the books. We observe that *price*, the name token the aggregation function attaching to, is related to *book* in different ways in the two queries. We also notice that the connection marker "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 as shown in Figure 7 to take the semantic contribution of connection markers into consideration.

We then propose the mapping rules as shown in Figure 8 to determine the nesting scope for aggregation functions. Specifically, we identify two different nesting scopes in a LET clause that results from using an aggregation function—*inner* and *outer*, with respect to the basic variable $\langle var \rangle$ that the function directly attaches to.

```
$CM \Rightarrow \langle connector \rangle$
$NT_1 \Rightarrow \langle var_1 \rangle$
$NT_2 \Rightarrow \langle var_2 \rangle$
$\langle function_1 \rangle + \langle function_2 \rangle + \langle var_2 \rangle \Rightarrow \langle cmpvar \rangle$
$\langle var_1 \rangle + \langle connector \rangle + \langle cmpvar \rangle \rightsquigarrow$
                              Create a new variable binding $NT_2 \Rightarrow \langle var_2 \rangle'$
                              if $\langle function_1 \rangle \neq$ null, then
                                 where $\langle function_2 \rangle + \langle var_2 \rangle' = \langle cmpvar \rangle$
                              else
                                 where $\langle var_2 \rangle' = \langle cmpvar \rangle$
Record $\langle var_2 \rangle'$ as related to $\langle var_1 \rangle$, $\langle var_2 \rangle$ as not related to $\langle var_1 \rangle$
```

Fig. 7. Semantic contribution of connection marker in query translation.

```
Given a non-core name token $NT$ (binding to $\langle var \rangle$), denote $NT_{core}$ (binding to $\langle core \rangle$)
as the core token related to $NT$, if any, else as a name token that $NT$ attaching
to and directly related to, if any; else as a randomly chosen name token that is not
directly related to $NT$.
Denote $\langle v \rangle$ as variables directly related to $\langle var \rangle$.

if given $\langle function \rangle + \langle var \rangle \Rightarrow \langle cmpvar \rangle$
  do $\langle cmpvar \rangle \rightsquigarrow$

  if $NT$ is not a core token itself, or there exists no core token, then

          let $\langle vars \rangle$ := {
             for $\langle core_1 \rangle$ in $\langle doc \rangle // NT_{core}$
             where $\langle core_1 \rangle$ = $\langle core \rangle$
             return $\langle var \rangle$}

       Replace $\langle cmpvar \rangle$ with $\langle function \rangle + \langle vars \rangle$.
       Mark $\langle var \rangle$ and $\langle core \rangle$, $\langle v \rangle$ and $\langle core \rangle$ as unrelated.
       Mark $\langle var \rangle$ and $\langle core_1 \rangle$, $\langle v \rangle$ and $\langle core_1 \rangle$ as related.
       Mark nesting scope for the LET clause as outer with respect to $\langle var \rangle$.

  else if $NT$ is a core token itself, or no $\langle core \rangle$ exists, then
          let $\langle vars \rangle$ := { return $\langle var \rangle$}
       Replace $\langle cmpvar \rangle$ with $\langle function \rangle + \langle vars \rangle$.
       Mark nesting scope for the LET clause as inner with respect to $\langle var \rangle$.

else if given $\langle function_1 \rangle + \langle function_2 \rangle + \langle var \rangle \Rightarrow \langle cmpvar \rangle$
        and $\langle function_2 \rangle + \langle var \rangle \Rightarrow \langle cmpvar_2 \rangle$
  then $\langle cmpvar \rangle \rightsquigarrow$
          let $\langle vars \rangle$ := {$\langle cmpvar_2 \rangle$}
  Recursively rewrite $\langle cmpvar_2 \rangle$.
  Replace $\langle cmpvar \rangle$ with $\langle function \rangle + \langle vars \rangle$.
```

Fig. 8. Grouping and nesting scope determination for aggregation functions.

If an aggregation function attaches to a basic variable $\langle var \rangle$ 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,

```
(1) count($v_2$)→$cv_1$
  $v_2$ is not a core token, and the core
token related to it is $v_1$, therefore
$cv_1$ ⇝
      let $vars_1$ := {
        for $v_5$ in ⟨doc⟩//director
        where   $v_5$ = $v_1$
        return   $v_2$ }
  Replace all $cv_1$ with count($vars_1$).
  Mark $v_2$, $v_1$ as unrelated.
  Mark $v_2$, $v_5$ as related.
  Mark nesting scope for the LET clause
as outer with respect to $v_2$.
```

```
(2) count($v_3$)→$cv_2$
  $v_3$ is not a core token, and the core
token related to it is $v_4$, therefore
$cv_1$ ⇝
      let $vars_2$ := {
        for $v_6$ in ⟨doc⟩//director
        where   $v_6$ = $v_4$
        return   $v_3$ }
  Replace all $cv_2$ with count($vars_2$).
  Mark $v_3$, $v_4$ as unrelated.
  Mark $v_3$, $v_6$ as related.
  Mark nesting scope for the LET clause
as outer with respect to $v_3$.
```

Fig. 9. Grouping and nesting scope determination in Query 2.

Table V. 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

the condition clause "where $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 the variable ⟨*var*⟩ (in this case $movie), indicating that in the XQuery construction phrase (Section 3.2.4), all the variables related to it should be put inside the LET clause.

On the other hand, when there exists no core token in the query, ⟨*var*⟩ may be associated with other variables indirectly related to it only via value joins. The nesting scope of the LET clause should be marked as *outer* with respect to ⟨*var*⟩. Similarly, if an aggregation function attaches to a basic variable ⟨*var*⟩ representing a noncore token, only those clauses containing variables directly related to ⟨*var*⟩ should be put inside the LET clause, because ⟨*var*⟩ 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 ⟨*var*⟩. In such a case, only those directly related to ⟨*var*⟩ should be put inside the LET clause during the full query construction phrase (Section 3.2.4). Query 2 in Figure 1 contains such cases, and the nesting scope determination is illustrated in Figure 9. The updated variable bindings and relationships between basic variables for the query are shown in Table V.

The nesting scope determination for a quantifier (Figure 10) is similar to that for an aggregation function, except that the nesting scope is now associated with a quantifier inside a WHERE clause. The nesting scope of a quantifier is marked

```
/*Both ⟨var⟩ and ⟨core⟩ are the same as that defined in Figure 8*/

if given ⟨quantifier⟩+⟨var⟩ ⇒ ⟨cmpvar⟩
  then ⟨cmpvar⟩ ⇝

  if ⟨var⟩ is not a core token itself, or there is no core token, then

            let ⟨vars⟩ := {
               for  ⟨core_1⟩ in  ⟨doc⟩//NT
               where  ⟨core_1⟩ = ⟨core⟩
               return  ⟨var⟩}
            where ⟨quantifier⟩ ⟨var_1⟩ in ⟨vars⟩ satisfies { }

       Mark ⟨var⟩ and ⟨core,⟩, ⟨core_1⟩ as unrelated.
       Replace ⟨var⟩ elsewhere with ⟨var_1⟩, 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⟩ ⟨var_1⟩ in ⟨vars⟩ satisfies { }

       Mark nesting scope for the WHERE clause with the quantifier as inner with
       respect to ⟨var⟩.
       Replace ⟨var⟩ elsewhere with ⟨var_1⟩, except in FOR clause.
```

Fig. 10. Grouping and nesting scope determination for quantifier.

as *inner* with respect to the variable ⟨*var*⟩ that the quantifier attaches to, when ⟨*var*⟩ is a core token. Otherwise, it is marked as *outer* with respect to ⟨*var*⟩.

3.2.3.1 ***mqf*** *Function.* As we have previously discussed in Section 3.2.1, 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 the **mqf** function can be obtained for each set of related basic variables:

```
let (⟨var_1⟩, ..., ⟨var_m⟩) be the union of all ⟨var⟩s related to each other
(⟨var_1⟩, ..., ⟨var_m⟩) ⇝  where mqf(⟨vars⟩)
```

For Query 2 in Figure 1, we can see that two sets of related variables can be found: {$\$v_2$,$\$v_5$} and {$\$v_3$,$\$v_6$}. The corresponding WHERE clauses containing **mqf** function are: where mqf($\$v_2$,$\$v_5$) 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.

According to the nesting scopes determined by the algorithms in Figures 8 and 10, we construct the query starting from innermost clauses and work outwards. If the scope defined is *inner* with respect to ⟨*var*⟩, then all the other query fragments containing ⟨*var*⟩ or basic variables related to ⟨*var*⟩ are put within an

```
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. 11. Full translation for Query 2.

inner query following the FLOWR convention (e.g., conditions in WHERE clauses are connected by and) as part of the query at the outer level. If the scope defined is *outer* with respect to ⟨*var*⟩, then only query fragments containing ⟨*var*⟩, and clauses (in case of quantifier, only WHERE clauses) containing basic variables directly related to ⟨*var*⟩ are put inside the inner query, while query fragments of other basic variables indirectly related to ⟨*var*⟩ 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.

As an example, the fully translated XQuery for Query 2 in Figure 1 is shown in Figure 11. We now describe the details of the query construction process. We begin with the two LET clauses obtained in Figure 9. The first LET clause is marked as *outer* with respect to v_2. Thus we put the FOR clause corresponding to v_2 (Table IV) inside the LET clause. We then examine basic variables that are related to v_2 and put them inside the same LET clause as v_2 (Table V). In this case, v_5, the only basic variable related to v_2, has already been included in the LET clause, so we need to do nothing. We then complete the LET clause by adding all the WHERE clause containing the variables in the clause. In this example, the only WHERE clause that can be added is "where mqf($\$v_2$,$\$v_5$)". Hence, we add the WHERE clause and complete the first LET clause. We similarly complete the second LET clause. After that, we add the FOR clause for the remaining basic variables v_1 and v_4, followed by the two LET clauses that we just have created. Finally, we add the remaining WHERE clauses ("where count($\$vars_1$) = count($\$vars_2$)" and "where $\$v_4$ = "Ron Howard") and RETURN clause ("return $\$v_1$") based on Tables IV and V according to the FLOWR convention. This step completes the query construction.

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 vocabulary 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 through precise querying.

To ensure that this process proceeds smoothly for the user, our system provides the user with helpful feedback on how to rephrase. In this section, we describe the validation process used to determine whether the system can translate a user query, as well as the informative error messages the system produces when validation fails.

NaLIX is designed to be a query interface for XML that translates natural language queries into Schema-Free XQuery. As such, the linguistic capability of the system is essentially constrained by the expressiveness of XQuery. A natural language sentence that can be understood and thus meaningfully mapped into XQuery by NaLIX is one whose semantics are expressible in XQuery. Furthermore, for the purpose of query evaluation, only the semantics that can be expressed by XQuery need to be extracted from the natural language sentence.

Consider the following query: “Find all the movies directed by director Ron Howard.” The semantic meaning of “directed by” cannot be directly expressed with XQuery. It is neither possible nor necessary for NaLIX to understand such semantics. Instead, based on the dependency parse tree of the query, the system 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 language query are composed of 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.

Table VI shows the grammar for the subset of natural language corresponding to XQuery semantics that is supported by NaLIX (ignoring all markers). We call a normalized parse tree that satisfies the grammar a *valid* parse tree.

A valid parse tree can be translated to an XQuery expression as described in Section 3.2. An invalid parse tree, however, will be rejected by the system, with error message(s).[6]

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 12. An error message will be returned to the user; *the same as* will be suggested as a possible replacement for *as*. If the user takes the suggestion,

[6]More details on the generation of error and warning messages in NaLIX can be found in the Electronic Appendix.

Table VI. Grammar Supported By NaLIX

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)

Symbol "+" represents attachment relation between two tokens; "[]" indicates implicit token, as defined in Definition 4.1

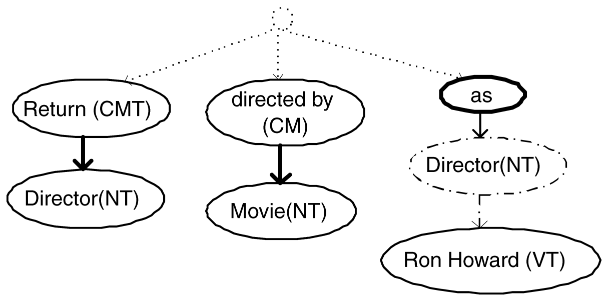


Fig. 12. Parse tree for Query 1 in Figure 1.

Query 2 in Figure 4 could be the new query issued by the user. 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. Although we generally assume user queries are written in grammatically correct English, no additional rules are needed to deal with incorrect English. If the parse tree of a grammatically incorrect English sentence is valid, the sentence can still be translated into XQuery.

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 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 alert the user of the potential misunderstanding.

During the validation process, we additionally perform the following procedures to deal with some data specific cases.

—*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 contained the database that represents 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 [Miller et al. 1990] and domain-specific ontology whenever one is available. The technique we currently use for ontology-based term expansion is similar to what has been previously described in Li et al. [2004, 2007c].

—*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 from Figure 12, 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 tokens* as defined below. See Table VI for the definitions of GVT, GOT, and RNP.

Definition 4.1 (*Implicit Name Token*). For any GVT, if it is not attached by a command token, nor adjacent to a RNP, nor attached by a GOT that is attached by an RNP or GVT, then each value token within the GVT is said to be related to an *implicit name token* (denoted as [NT]). An implicit name token related to a value token is the name(s) of element or attribute with the value of a value token in the database.

We determine the implicit name token associated with a value token by looking up the names of elements/attributes with the given value in the database. If the names matching a name token or the value of a value token cannot be found in the database, an error message will be returned. If multiple elements or attributes with different names matching the name token or value token are found in the database, the disjunction 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. SUPPORT ITERATIVE SEARCH

Previous studies [Olson et al. 1985; Lauer et al. 1992; Moore 1995] have shown that search is rarely a single-step process. Users often iteratively modify their queries based on the results obtained. It is thus important to provide a system to support a sequence of related queries. In this area of functionality, NaLIX keeps a query history and query template mechanism to allow old queries to be easily located, refined, and resubmitted. But this is not enough in itself—in normal human discourse, followup queries are often only partially specified. Forcing the user to fully specify each followup query gets in the way of normal iterative search. Furthermore, sometimes followup queries are required to express complex query semantics with ease in a divide-and-conquer fashion, especially when the semantics are too complex to be composed comfortably into a single query sentence. In this section, we describe how NaLIX supports followup queries, thereby allowing users to incrementally focus their search on the objects of interest while being able to back up at any time, and return to any point of recent search history should they decide to take a different search direction.

Q4. *Return the titles for all the movies directed by Ron Howard after 1990.*
- Q4.1. *Only for the movies made after 1995 but before 2000.*
  - Q4.1.1. *Return the actors for each movie as well.*
  - Q4.1.2. *Only for those made in 1998.*
- Q4.2. *How about those directed by Peter Jackson?*
  - Q4.2.1. *Return their producers as well.*

Q5. *Find the author of book "Gone with the Wind."*
- Q5.1. *Find the publisher for it as well.*
- Q5.2. *Find all the books by her.*

Fig. 13. Example query threads.

5.1 Basic Model

Iterative search by issuing followup queries could be considered analogous to conversation in a "chat room" with only two participates—a user and NaLIX—where the user issues questions, and NaLIX responds to each question by querying the database. However, previous studies [Hegngi 1998; Vronay et al. 1999; Smith et al. 2000] have pointed out a few serious drawbacks of such "chat room" style communication, including the lack of context and uselessness of chat history, all attributed to the unorganized text flows. Meanwhile, forum-style communication is praised for its structured history and ease of context retrieval [Hegngi 1998; Farnham et al. 2000; Sheard et al. 2003]. Unfortunately, unlike a chat room, a forum is not designed for synchronous communication such as database search and thus cannot be directly applied to iterative search. To get the best from both worlds, we design the following model of iterative search for our system: while the interaction between a user and the system for each individual query remains synchronous as in a chat, queries are explicitly organized into forum style topic threads so that the context of each query is well maintained.

In our model, the basic unit of iterative search is a *query tree*. Each query tree is composed of multiple queries on a single topic or multiple related topics. The root of a query tree is the first query submitted by the user to initiate search regarding a specific topic (e.g., Q4 and Q5 in Figure 13), referred as a *root query*. The query tree then expands as the user submits new queries to refine existing queries in the query tree (e.g., Q4.1 and Q4.1.1 in Figure 13). When the user submits a followup query Q_c to an existing query Q_p, we will then record Q_p as the *parent query* of child Q_c in the query tree, and Q_c is called a *child query* of Q_p. A parent query Q_p can be a root query or a followup query itself, but a child query Q_c is always a followup query. Multiple followup queries can be issued for the same query so that Q_p can have additional children. We also expect that a user will explicitly specify the parent for each followup query. If no parent is specified, we treat the given query as a root query and create a new query tree. Hence, with our model, although iterative search appears as series of queries, these queries are organized explicitly into tree-hierarchies. A user's previous search effort is preserved within context; the user can easily return to any point of recent search history to take a different search direction.

5.2 The Main Challenges

The system described so far only handles root queries. To support followup queries, the fundamental challenge is how to identify the semantic meaning of a partially specified followup query (such as Q4.1 in Figure 13) in the light of its prior queries. To address this challenge, we first need to identify what has been inherited by a followup query from its prior queries. To do so, the issues we need to solve are: (i) identification of equivalent objects between a followup query and its prior queries; and (ii) reference resolution to determine the semantic meaning of references to prior queries in the followup query. For example, to translate Q5.1 in Figure 13, we need to determine the meaning of *it*—whether *it* refers to *author* or *book* in the parent query Q5. In addition, since a followup query is used to refine a prior query, we also need to determine all the modifications that need to be done on the prior query. The types of modifications we consider include *addition* (e.g., adding new constraints), *substitution* (e.g., replacing a selection predicate), and *topic switch* (e.g., start asking about *book* instead of *movie*). Finally, limited linguistic capability remains an issue when handling followup queries; we thus need to design interactive facilities to guide users to formulate followup queries.

The rest of this section focuses on our solution to the above challenges. Section 5.3 defines basic notions related to followup queries. Section 5.4 describes how a followup query is tokenized and translated into an XQuery expression, with a focus on issues unique to followup queries. Before query translation, a followup query needs to be validated and has its references resolved, but we defer discussion of these two steps to Section 5.6 and Section 5.5, respectively. Finally, in Section 5.7 we propose an extension based on the notions in Section 5.3 to automatically determine root queries that are mistakenly submitted as followup queries by the user. A sketch of the translation process that supports followup queries is presented in Figure 14.

5.3 Query Context

A root query starts a query tree and defines the basic topic the query tree is focused on. Followup queries are then used to refine the query topic. We refer to the information contained in an existing query as *query context*.

Definition 5.1 (*Query Context*). *Query context* of a query refers to tokens in the query and the patterns of tokens that correspond to XQuery fragments, and the query context of its parent query, if one exists.

In natural language processing, *centering theory* claims that a discourse always has a distinguished discourse entity that represents the topic of the discourse [Grosz et al. 1986, 1995]. Similarly, a query tree always has a distinguished object of interest that represents the query topic. Such an object of interest is usually specified along with other related objects in a query. These related objects are either explicitly constrained (e.g., *in year 2000*) or restricted by the object of interest in the form of dependent relations (e.g., *title of all the movies*). Meanwhile, the object of interest itself is rarely directly restricted with any specific conditions. For instance, given a query "Find the titles of all the

```
1. Given a natural language query $S_Q$
2. Obtain parse tree $T_Q$ for $S_Q$ from a dependency parser
3. if $S_Q$ is a not a followup query
Step 1$_a$:
4.    Classify words/phrases in $T_Q$ into tokens/markers based on Table I and II
Step 2$_a$:
5.    Validate $T_Q$ based on the grammar supported for stand-alone queries (Table VI)
Step 3$_a$:
6.    if $T_Q$ is valid then
7.       Determine context center $T_Q.CtxCenter$
8.       Assign variables for name tokens in $T_Q$
9.       Map parse tree fragments in $T_Q$ into XQuery fragments
10.      Determine groupings and nestings
11.      Add the XQuery fragments and their grouping/nesting information to $T_Q.Ctx$
12.      Construct full XQuery expression $Q$ based on query context $T_Q.Ctx$
13.      return $Q$ and MessageGenerator.Warnings($T_Q$)
14.    else
15.      return MessageGenerator.Errors($T_Q$)
16. else
17.    let $P_Q$ be the parse tree for the parent query of $T_Q$
Step 1$_b$:
18.    Classify words/phrases in $T_Q$ into tokens/markers based on Table I,II,and VII
Step 2$_b$:
19.    Validate $T_Q$ based on the grammar supported for followup queries (Table XIII)
Step 3$_b$:
20.    if $T_Q$ is valid then
21.       $T_Q.Ctx$ := $P_Q.Ctx$
22.       Determine context center $T_Q.CtxCenter$
23.       Assign variables for name tokens and reference tokens in $T_Q$
24.       Map parse tree fragments in $T_Q$ into XQuery fragments
25.       Determine groupings and nestings
26.       Updated $T_Q.Ctx$ based on the new XQuery fragments and their group/nesting
27.       Construct full XQuery expression $Q$ based on $T_Q.Ctx$
28.       return $Q$ and MessageGenerator.Warnings($T_Q$)
29.    else
30.       return MessageGenerator.Errors($T_Q$)
```

Fig. 14. Translation process overview: from natural language query to XQuery with support for followup queries.

movies directed by John Howard in year 2006," we can determine that the main topic of the query is *movie*s for the following reasons. First, the objects *director* (corresponding to *John Howard*) and *year* are simply used to add constraints on *movie*s, with their corresponding variables included in WHERE clauses. Second, in the dependency parse tree *movies* is below *titles*, indicating that *titles* is restricted by *movies*. In some cases, the object of interest itself may be constrained. For example, one can write "Find all the titles containing *Wind*." But in such cases, the user is also interested to see the object of interest in the results (corresponding variable is included in a RETURN clause). To capture the above intuition, we define the following notion of *context center* to represent the object of interest in a query.

Definition 5.2 (*Context Center*). A *context center* is the lowest name token among those whose corresponding basic variables are not included in a WHERE clause. If no such name token exists, then a *context center* is a name token whose corresponding basic variable is included in a RETURN clause.

When a query contains core tokens, the context center of the query must be a core token, as a core token is always at the leaf level of a parse tree according to Definition 3.2. If there are multiple core tokens, we pick the first core token as the context center, as other core tokens are typically used to specify constraints on the first one in the form of value join. For example, the context center for Q2 is *director* (node 7 in Figure 3), which is the first core token of the query; the other core token (node 11) is not the context center.

A followup query inherits and modifies the query context of its parent query (referred as *parent query context*) to create its own query context. It usually inherits the context center as well. For example, Q4 specifies the topic of interest to be *movie*s made by a particular *director* after certain *year*; its followup query Q4.1 imposes more restrictions over *year* but is also looking for *movie*s. A followup query can be partially specified and contains no context center. For example, the user can specify "But before 2000" as a followup query to Q4 in Figure 13. The only name token *year* is not a context center according to Definition 5.2, as it only appears in a WHERE clause. In such a case, the query simply inherits the context center of its parent query. A followup query can also change the context center. For instance, in Figure 13, Q5.1 changes the context center from *author* in Q5 to *publisher*. Different context centers in the same query tree may simply be viewed as disjunctive objects of interest to the user. For ease of discussion, in this article we limit a query tree to have only one context center at any time. But it is straightforward (by considering disjunctive variables) to extend our techniques to allow multiple context centers.

5.4 Query Translation for Followup Query

Translation for followup queries involves the same three major steps for standalone or root queries as described in Section 3, namely, token classification, parse tree, validation, and query translation. In this section, we focus on issues unique to followup queries. Techniques discussed in Section 3 are generally applicable to followup queries unless noted otherwise.

5.4.1 *Token Classification.* To translate a followup query into an XQuery expression, we first need to identify words/phrases in the original sentence that can be mapped into corresponding XQuery components as described in Section 3.1. In addition, we also need to identify words/phrases[7] (Table VII)

[7]A pronoun that can be classified as Reference Token is no longer classified as Pronoun Marker.

Table VII. New Types of Tokens and Markers

Type of Token	Query Component	Description	Examples
Reference Token (RT)	Basic Variables	A word from an enum set of pronouns and possessive adjectives	*those*, *her*

Type of Marker	Semantic Contribution	Description	Examples
Substitution Marker (SM)	Indication for results/sorting specification replacement	A word from an enum set of adverbs	*instead*, *only*

Table VIII. Variable Bindings for Query 4

Variable	Associated Content	Related To
$\$v_1$	*title*	$\$v_2$, $\$v_3$, $\$v_4$
$\$v_2$[⊛]	*movie*	$\$v_1$, $\$v_3$, $\$v_4$
$\$v_3$	*director*	$\$v_1$, $\$v_2$, $\$v_4$
$\$v_4$	*year*	$\$v_1$, $\$v_2$, $\$v_3$

⊛: corresponding name token is context center.

representing references to the prior queries, which are essential to the query semantics of the followup query. The classified parse tree is then validated, but we defer the discussions of parse tree validation for followup queries to Section 5.6.

5.4.2 *Translation into XQuery*. Given a valid followup query, we can translate it into XQuery. The major challenge here is to construct an XQuery expression given a followup query and its ancestor queries in the query tree.

5.4.3 *Core Token Identification and Variable Binding*. The core token identification and variable binding for a followup query is essentially the same as that for a root query, with the following key difference. A name token NT_c in a followup query is bound to a new basic variable, unless it is regarded as identical to a name token NT_p in its parent query context. In such a case, the name token NT_p is called an *inherited name token* of NT_p and is assigned to the same variable as NT_p (say, $\$v_p$). The list of related variables for $\$v_p$ is also updated based on the relationships of tokens in the followup query.

Definition 5.3 (*Inherited Name Token*). A NT_c in a followup query is referred to as an inherited name token of NT_p in its parent query context, if and only if (i) they are equivalent,[8] (ii) they are attached by the same name token or its anaphora,[9] and (iii) no FT or QT attaches to either of them.

Tables VIII and IX show the variable bindings for Q4 and Q4.1 in Figure 13, respectively. As can be seen, the name token *movie* in Q4.1 is assigned to the

[8]Based on Definition 3.1, or if NT_c is not implicit but NT_p is, they are still considered as equivalent if they are composed of the same noun phrase and directly related to equivalent NTs.
[9]An anaphora of a name token is a pronoun that refers back to the name token.

Table IX. Variable Bindings for Q4.1

Variable	Associated Content	Related To
$\$v_2^{\circledast}$	*movie*	$\$v_1$, $\$v_3$, $\$v_4$
$\$v_4$	*year*	$\$v_1$, $\$v_2$, $\$v_3$

Table X. Direct Mapping for Query 4.1

Pattern	Query Fragment
$\$v_2$	for $\$v_2$ in $\langle doc\rangle$//movie
$\$v_4$	for $\$v_3$ in $\langle doc\rangle$//year
$\$v_4 + > + \langle constant\rangle$	where $\$v_4 > 1995$
$\$v_4 + < + \langle constant\rangle$	where $\$v_4 < 2000$

same variable $\$v_2$ as the name token *movie* in Q4, since it is an inherited name token of the latter.

During the variable binding process, we also assign variables for reference tokens. We defer further discussions of this topic to Section 5.5.

5.4.4 *Mapping.* Patterns of tokens in a followup query can be mapped into XQuery fragments as described in Section 3.2.2. Tables X and XI show a list of direct mappings from token patterns to query fragments for Q4 and Q4.1 in Figure 13, respectively. The determination of grouping and nesting of XQuery fragments is the same as described in Section 3.2.3.

5.4.5 *Update Query Context.* The purpose of a followup query is to refine its parent query by modifying the query context inherited from its parent query. Three types of revisions to the parent query context can be specified in a followup query:

—*Type (a): Addition.* A followup query can be used to add new constraints and/or results/sorting specifications to its inherited query context. The patterns for the additional information are simply added to query context and used for XQuery construction later on.

—*Type (b): Substitution.* A followup query can also be used to specify constraints and results/sorting specifications that replace existing conditions and results/sorting specifications in the query context of its parent query. Corresponding patterns in the original query context are then substituted by the new patterns.

The algorithm in Figure 15 implements the above updates to query context for a followup query. For example, as shown in Table XII, Q4.1 in Figure 13 modifies the constraint over $\$v_4$ (*year*) from "$\$v_4 > 1990$" in Q4 to "$\$v_4 > 1995$," and adds one more constraint "$\$v_4 < 2000$."

In addition, whenever the context center is changed in a followup query, modifications to the query context are also involved. As discussed in Section 5.1, we currently limit a query tree to have only one context center at a time.

—*Type (c): Topic Switch.* Whenever a context center is replaced by a new one in a followup query, any query fragment in the inherited query context that contains variables unrelated to the new context center is removed from the query context.

Table XI. Direct Mapping for Query 4

Pattern	Query Fragment
$\$v_1$	for $\$v_1$ in $\langle doc\rangle$//title
$\$v_2$	for $\$v_2$ in $\langle doc\rangle$//movie
$\$v_3$	for $\$v_3$ in $\langle doc\rangle$//director
$\$v_4$	for $\$v_4$ in $\langle doc\rangle$//year
$\$v_3 + \langle constant\rangle$	where $\$v_3$ = "Ron Howard"
$\$v_4 + > + \langle constant\rangle$	where $\$v_4 > 1990$
$\langle return\rangle + \$v_1$	return $\$v_1$

```
Let $\mathcal{C}$ denotes query context; subscript $_p$ stands for ``from parent query context,''
and $_c$ for ``from current query context.'' Operators within the same group $\{\leq,<,=\}$,
$\{\geq,>,=\}$ and $\{\neq\}$ are incompatible with each other.'' Let $\langle replace\rangle \rightarrow SM|\varnothing$.

WHERE clause:
  for $\langle variable_c\rangle + \langle opr_c\rangle + \langle arg_c\rangle$
        if $\exists$ $\langle variable_p\rangle = \langle variable_c\rangle$ and $\langle variable_p\rangle + \langle opr_p\rangle + \langle arg_p\rangle$, then
           if $\langle opr_p\rangle$ is incompatible with $\langle opr_c\rangle$, then
                remove $\langle variable_p\rangle + \langle opr_p\rangle + \langle arg_p\rangle$ from $\mathcal{C}$
           add $\langle variable_c\rangle + \langle opr_c\rangle + \langle arg_c\rangle$ to $\mathcal{C}$
ORDERBY clause:
  for $\langle sort_c\rangle + \langle replace_l\rangle + \langle variable_c\rangle + \langle replace_r\rangle \rightsquigarrow$ orderby $(\langle variable_c\rangle)$
        if $\langle replace_l\rangle \neq \varnothing$ or $\langle replace_r\rangle \neq \varnothing$, then
              remove $\langle sort\rangle + \langle variable_p\rangle$ from $\mathcal{C}$
        add $\langle sort\rangle + \langle variable_c\rangle$ to $\mathcal{C}$
RETURN clause:
  for $\langle cmd_c\rangle + \langle replace_l\rangle + \langle variable_c\rangle + \langle replace_r\rangle \rightsquigarrow$ return $(\langle variable_c\rangle)$
        if $\langle replace_l\rangle \neq \varnothing$ or $\langle replace_r\rangle \neq \varnothing$, then
              remove $\langle return\rangle + \langle variable_p\rangle$ from $\mathcal{C}$
        add $\langle return\rangle + \langle variable_c\rangle$ to $\mathcal{C}$
```

Fig. 15. Updating query fragments in query context.

For instance, after the context center is changed to *publisher* in Q5.1, the only pattern inherited from the parent query context is "$\$v_1$ + *Gone with the wind*," where $\$v_1$ refers to *book*, and the other pattern "$\langle return\rangle + \$v_2$," where $\$v_2$ refers to the original context center *author*, is removed from the query context of Q5.1.

A full XQuery expression can then be constructed based on the XQuery fragments contained by the updated query context. The construction process is the same as described in Section 3.2.4.

5.5 Reference Resolution

Reference resolution is an important step in query translation for followup queries, where semantic meanings of references to prior queries are identified. Ideally, we would like to directly take advantage of existing reference resolution tools. Unfortunately, all the tools available to us, such as LingPipe [Alias-i 2006] or GATE [Cunningham et al. 2002], are mainly designed to handle references

Table XII. Updated Query Context for Query 4.1

Pattern	Query Fragment
$\$v_1$	<u>for</u> $\$v_1$ <u>in</u> $\langle doc\rangle$//title
$\$v_2$	<u>for</u> $\$v_2$ <u>in</u> $\langle doc\rangle$//movie
$\$v_3$	<u>for</u> $\$v_3$ <u>in</u> $\langle doc\rangle$//director
$\$v_4$	<u>for</u> $\$v_4$ <u>in</u> $\langle doc\rangle$//year
$\$v_3 + \langle constant\rangle$	<u>where</u> $\$v_3$ = "Ron Howard"
$\$v_4 + > + \langle constant\rangle$	<u>where</u> $\$v_4 > 1995^{\dagger}$
$\$v_4 + < + \langle constant\rangle$	<u>where</u> $\$v_4 < 2000^{\ddagger}$
$\langle return\rangle + \$v_1$	<u>return</u> $\$v_1$

†The fragment is newly modified.
‡The fragment is newly added.

for named entities (proper nouns), such as *he* referring to *Ron Howard*, but not to contend with references to entities in the form of common nouns, such as *he* referring to *director*. Although the former type of reference is common in news corpora, the default training data for those tools, the latter is far more important for database queries. For example, for Q4.2 in Figure 13, we need to know that *those* refers to *movies* to be able to translate the query into a correct XQuery expression. In addition, existing tools cannot directly take advantage of semantic knowledge the system already has (e.g., *John Howard* is a director). Because of this, we have been compelled to derive new reference resolution techniques in NaLIX, which we now describe.

Our approach is similar to existing salience-based approaches [Mitkov 1998; Dimitrov et al. 2002], but has a more focused goal: we only seek to address resolution of pronoun anaphora between sentences where the antecedent is a common noun. To achieve this goal, we first define a new token type called *Reference Token (RT)* (Table VII). Reference resolution in NaLIX is therefore equivalent to the task of finding the corresponding name token(s) in the parent query context for a reference token.

Figure 16 ($\dashrightarrow$ is the abbreviation for *refer to*) presents our light-weight reference resolution approach. As can be seen, a reference token may refer to multiple antecedents in RETURN clause (e.g., *those* may refers to both *title* and *year*). In addition, since the context center is more likely to be referred to by followup queries, we give higher priority to the context center. For example, based on our algorithm, *those* in Q4.2 (Figure 13) refers to *movies* instead of *titles*. For others, we find the antecedent by just simply relying on number and gender[10] matches. Following this simple rule, we can find *it* in Q5.1 refers to *book* (implicit name token corresponding to *Gone with the Wind*) instead of *author* in Q5, while *her* in Q5.2 refers to *author*.

The concept of query context inheritance allows our system to be relatively robust against errors in reference resolution—in fact, reference resolution errors have no negative impact on query translation results, unless the reference token is involved in the situations that cause changes to query context as described in Section 5.4.2. For example, if Q4.1 is rewritten into "Only for those

[10]The gender for a given name token is neuter if *person* is not its hypernym according to WordNet; we do not distinguish between female and male.

```
Let getNum() be a function that returns the number of a word (single/plural), given
the word itself or its corresponding base variable.
Let getGen() be a function that returns the gender (neuter or not) of a word, given
the word itself or its corresponding base.
Denotes $\langle context \rangle$ as the context center of the parent query

$\langle ref \rangle \rightarrow$ RT
let $\langle ref \rangle \dashrightarrow \varnothing$
if $\langle CMT \rangle + \langle ref \rangle$ and $\langle ref \rangle$ is a pronoun
   then if $\exists$ return + $\langle variable_p \rangle$
         then $\langle ref \rangle \dashrightarrow \langle ref \rangle | \langle variable_p \rangle$
else if getNum($\langle ref \rangle$)=getNum($\langle context \rangle$) and getGen($\langle ref \rangle$)=getGen($\langle context \rangle$)
         then $\langle ref \rangle \dashrightarrow \langle context \rangle$
else if getNum($\langle ref \rangle$)=getNum($\langle variable_p \rangle$) and getGen($\langle ref \rangle$)=getGen($\langle variable_p \rangle$)
         then $\langle ref \rangle \dashrightarrow \langle variable_p \rangle$
```

Fig. 16. Reference resolution.

made after 1995 but before 2000," the resulting XQuery expression remains the same, even if *those* was wrongly determined as referring to *titles* instead of *movies*.

We realize that our solution, like any other existing reference resolution technique, is not perfect. Instead of frustrating users with unexpected wrong results, we actively seek users' help via interactive dialog, as described in the next section.

5.6 Interactive Query Formulation for Followup Query

A valid root query can be translated into a complete XQuery expression. Similarly, a valid followup query can also be translated into a complete XQuery expression. But unlike a root query, a followup query is often an incomplete (elliptical) sentence, whose meaning is complemented by its prior queries. A followup query is translated into a new XQuery expression based on its inherited query context and the refinements to the query context specified by the followup query. Since only a valid query is allowed to have followup queries, the parent query of any followup query is valid. In other words, the inherited query context of any followup query can always be mapped into a complete XQuery expression. Therefore, a valid followup query is still confined by XQuery syntax but only needs to provide valid XQuery fragments, instead of a valid full XQuery expression. The corresponding grammar for followup queries is specified in Table XIII. Note that a valid followup query must contain at least one RETURN, WHERE, or ORDERBY clause. In other words, it cannot be empty or contain only NT/RT tokens and/or markers. For the purpose of presentation, this restriction is not included in the grammar presented in Table XIII.

A valid parse tree of a followup query can be translated to an XQuery expression as described in Section 5.4. For an invalid parse tree, error messages will be generated based on the grammar in Table XIII in the same way as presented in Section 4 for root queries. Similarly, warnings will be generated when

Table XIII. Grammar Supported by NaLIX for Followup Queries

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.	|RT
8.	ORDER_BY → OBT+RNP
9.	RNP → RT | NT | PM |(QT+RNP)|(FT+RNP)|(RNP∧RNP)
10.	GOT → OT|(NEG+OT)|(GOT∧GOT)
11.	GVT → VT |(GVT∧GVT)
12.	CM → (CM+CM)

Symbol "+" represents attachment relation between two tokens; "[]" indicates implicit token, as defined in Definition 4.1.

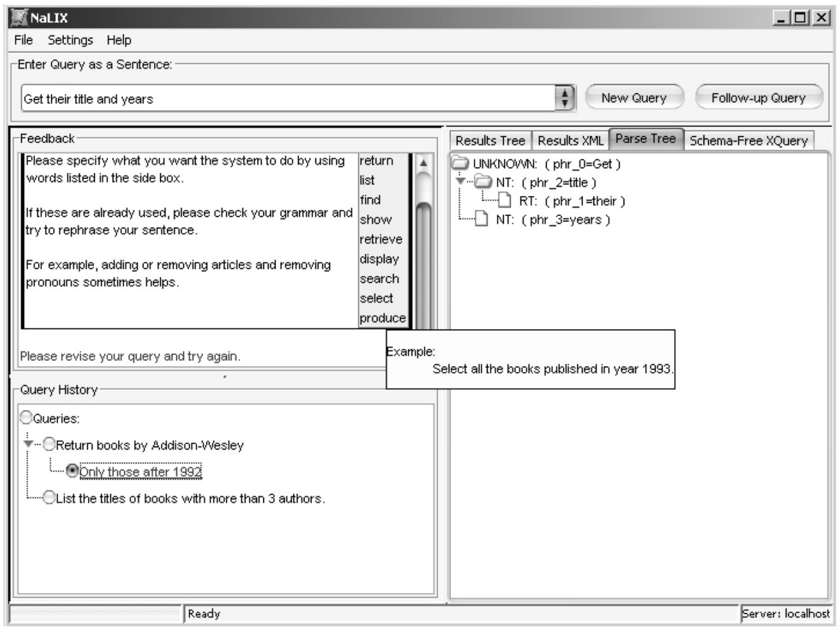
the system is not able to correctly interpret a valid followup query, including situations that are unique to followup queries such as ambiguity in reference resolution. A subtle yet important change in the warning handling for followup queries is that if, a followup query leads to the same warning message as its parent query, this warning message is suppressed. This change is based on our assumption that if a user has already chosen to ignore a warning message (by typing a new query causing the same warning), then the same warning message is likely to be ignored again. If so, displaying the same "useless" message again is more likely to be annoying instead of helpful to the user.

Figure 17 illustrates an example iteration in NaLIX, in which the user successfully formulate a valid followup query with the help from the system.

5.7 Query Tree Detection

Keeping track of query trees is important for supporting iterative search. The notion of query tree not only helps the system to organize queries in a meaningful way, but also allows the system to handle followup queries within a finite scope. In the base system we described so far, we expect a user to explicitly specify a query as root query to create a new query tree. Unfortunately, we found that users often simply type in a query as a followup query even though the query is self-contained and essentially starts a new query tree. Such situations also occur in email communications and online forums, where discussions totally irrelevant to the original topic may be carried on under the same subject heading, complicating the task of navigating through messages. Similarly, such "fake" followup queries in our system can make it difficult for users to navigate query trees and comprehend query context. In such cases, the usefulness of a query tree is greatly undermined. Hence, we propose the following extension to automatically detect such queries and identify them as the start of a new query tree.

The basic idea is that we can detect new query trees by finding followup queries that are in fact root queries. The main difference between a root query and a followup query is that a root query contains all the information needed to



Get their title and years

Please specify what you want the system to do by using words listed in the side box.

If these are already used, please check your grammar and try to rephrase your sentence.

For example, adding or removing articles and removing pronouns sometimes helps.

return
list
find
show
retrieve
display
search
select
produce

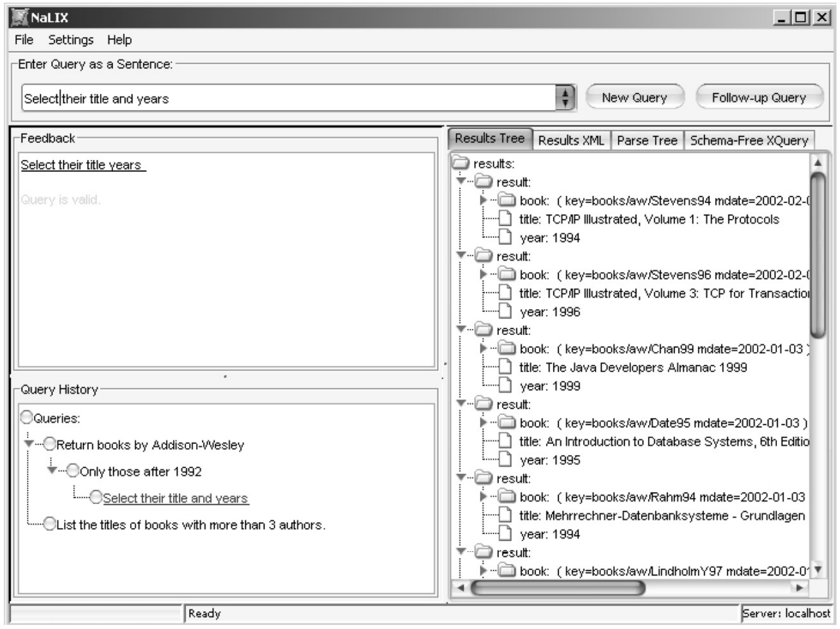
Please revise your query and try again.

Example:
Select all the books published in year 1993.

```
UNKNOWN: ( phr_0=Get )
NT: ( phr_2=title )
    RT: ( phr_1=their )
NT: ( phr_3=years )
```

```
Queries:
  Return books by Addison-Wesley
    Only those after 1992
  List the titles of books with more than 3 authors.
```

(a)



Select their title and years

Select their title years

Query is valid.

```
Queries:
  Return books by Addison-Wesley
    Only those after 1992
      Select their title and years
  List the titles of books with more than 3 authors.
```

```
results:
  result:
    book: ( key=books/aw/Stevens94 mdate=2002-02-0
    title: TCP/IP Illustrated, Volume 1: The Protocols
    year: 1994
  result:
    book: ( key=books/aw/Stevens96 mdate=2002-02-0
    title: TCP/IP Illustrated, Volume 3: TCP for Transactio
    year: 1996
  result:
    book: ( key=books/aw/Chan99 mdate=2002-01-03 )
    title: The Java Developers Almanac 1999
    year: 1999
  result:
    book: ( key=books/aw/Date95 mdate=2002-01-03 )
    title: An Introduction to Database Systems, 6th Editio
    year: 1995
  result:
    book: ( key=books/aw/Rahm94 mdate=2002-01-03
    title: Mehrrechner-Datenbanksysteme - Grundlagen
    year: 1994
  result:
    book: ( key=books/aw/LindholmY97 mdate=2002-0
```

(b)

Fig. 17. An example iteration: (a) NaLIX generates an error message for an invalid followup query. The user moves the mouse over a suggested word to see its example usage. The selected query in Query History provides query context. (b) NaLIX displays search results after the user corrects the query.

construct an XQuery expression, whereas a followup query needs to inherit and modify the query context of its parent to create an XQuery expression. Therefore, if a given query entered as a followup query does not need any information provided by its prior queries, then it can be regarded as a new root query. We now formalize this idea based on the notions defined in Section 5.1.

Definition 5.4 (*New Query Tree*). A query is *a root query of a new query tree*, if (i) it has no parent query, or (ii) its context center is different from the context center of its parent query, and all the pattern of tokens in the parent query context can be removed.

Based on the above idea, whenever a user submits a new query as a followup query, we can automatically determine whether this given query is in fact a root query; if it is, a new query tree with this query as root query is automatically created. For example, if Q5 in Figure 13 is typed in as a followup query for Q4, it will be automatically determined to be a new root query for the following reasons. First, the context center is changed from *movie* in Q4 to *author* in Q5. Second, Q5 contains no inherited name token; all the query fragments in the query context inherited by Q5 thus can be removed. In another word, Q5 does not need any information from its prior queries to construct an XQuery expression, and thus is in fact a root query.

6. EXPERIMENT

We implemented NaLIX as a stand-alone interface to the TIMBER native XML database [Jagadish et al. 2002] that supports Schema-Free XQuery. 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 [Schmidt et al. 2001]. 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.

6.1 Methods

Participants were recruited with flyers posted on a university campus. Eighteen of them completed the full experiment. Their ages 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.

6.1.1 *Procedures.* The experiment was a within-subject design. Each participant completed two experimental blocks. In an experimental block, a participant used either NaLIX or a keyword search system to accomplish nine search tasks in a random order determined by a pair of orthogonal 9 × 9 Latin squares (The order of using the two systems was therefore also counterbalanced for all participants.)

The search tasks were adapted from the "XMP" set in the XQuery Use Cases [World Wide Web Consortium 2003]. Each search task was described with the elaborated form of an "XMP" query[11] taken from XQuery Use Cases [World Wide Web Consortium 2003]. 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 [Shaw et al. 1997] greater than 0.5 was set as the 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 min was set for each task. If a participant reached the criteria before the time limit, he/she was given the choice to move on or to revise the query to get better results.

6.1.2 *Measurement.* We evaluated our system on two 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 three out of four attributes requested for each element, would have a recall score of 75%. The ordering of results was not considered when computing precision and recall, unless the task specifically asked the results be sorted.

[11]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.

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 1 s), 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 subcollection 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 1.44 MB, with 73,142 nodes when loaded into TIMBER. We chose DBLP because it is semantically close to the data set coming with XMP user 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 article, we felt that it is most appropriate to use this data set to balance the tradeoff between performance and realism: we minimized the overhead resulted from using 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 was 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.

6.2 Results and Discussion

6.2.1 *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 18. As can be seen, the average total time needed for each search task is usually less than 90 s, 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 s, 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 the search tasks (not the same tasks for different 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 the 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

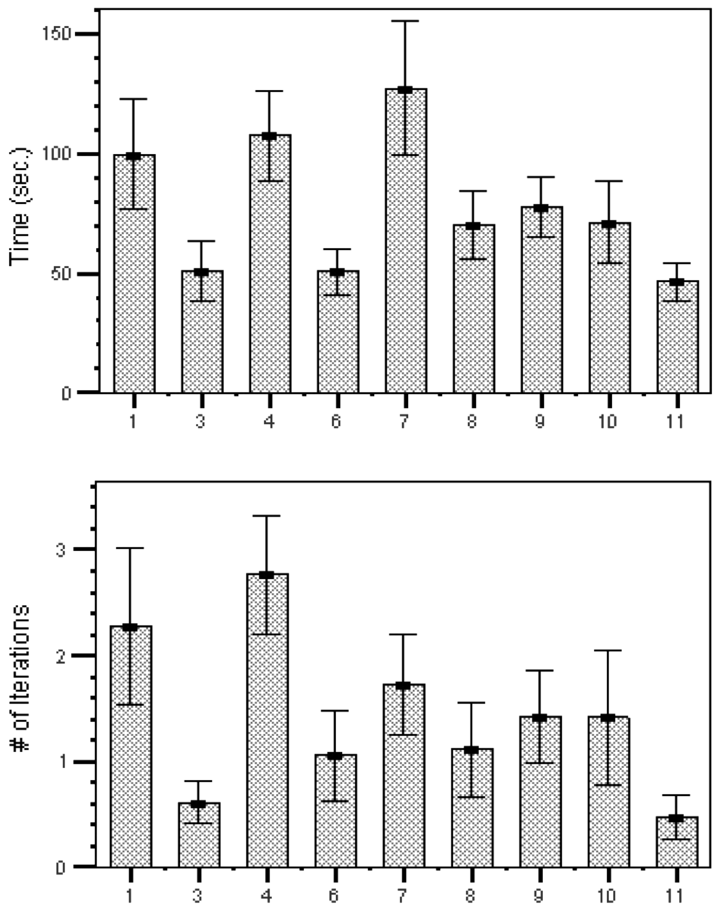
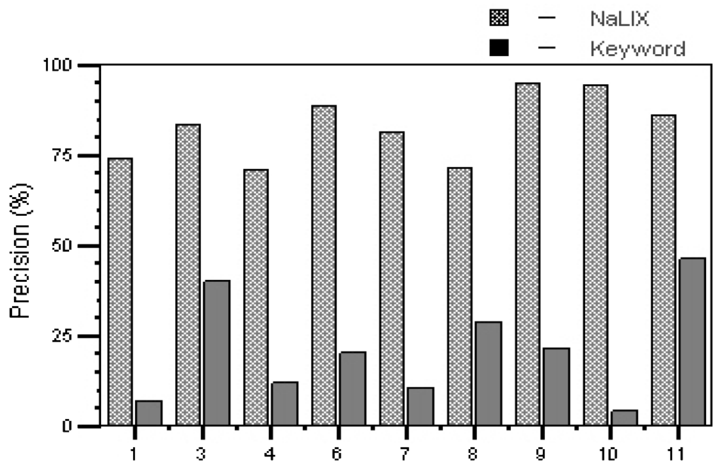


Fig. 18. Average time (in seconds) and average number of iterations needed for each "XMP" search task. Error bars show standard errors of means.

to use. The average level of satisfaction with NaLIX was 4.11 on a scale of 1 to 5, where 5 denotes *extremely easy to use*.

6.2.2 *Search Quality.* Figure 19 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 two out of the nine 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,[12] especially on those requiring complex manipulations such as aggregation or sorting (e.g., Q7, Q10). Even for queries with simple constant search conditions and

[12]Each search task corresponds to an "XMP" query in [World Wide Web Consortium 2003] with the same task number.



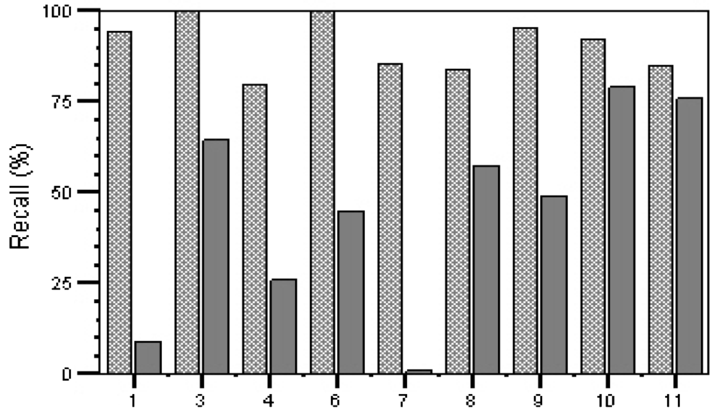


Fig. 19. Average precision and recall for each "XMP" search task.

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 books), resulting in a loss in 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.[13] 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*.

[13]MINIPAR achieves about 88% precision and 80% recall with respect to dependency relation with the SUSANNE Corpus [Lin 1998].

Table XIV. 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

Consequently, NaLIX failed to return *year* elements in the result, resulting in a loss in both precision and recall. Table XIV 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 this experiment.

7. RELATED WORK

In our preliminary version of this article [Li et al. 2006], we presented the basic algorithms on constructing a natural language interface for an XML database. A prototype of the NaLIX system has also been demonstrated at SIGMOD 2005 [Li et al. 2005]. Techniques for enabling domain-awareness of the NaLIX system have been developed Li et al. [2007b] and demonstrated at SIGMOD 2007 Li et al. [2007a]. These earlier works considers only a single lookup search model (see Section 3), which assumes that a user already has a specific well-defined search query in mind. In this work, we extend Li et al. [2006] to support iterative search, where a user can iteratively modify queries based on the results obtained. Specifically, we introduce the concept of followup query and a basic model for iterative search to maintain the sequence of queries in a dialog (Sections 5.1). Furthermore, Li et al. [2006] focused on only queries that are fully specified, whereas we also deal with queries that are only partially specified. The main challenge is to identify the semantic meaning of a partially specified query in the light of its prior queries. We first define the basic notion of query context (Section 5.3). We then extend the algorithm in Li et al. [2006] to support the translation from a followup query into an XQuery expression based on the query itself and its prior queries (Section 5.4). We also propose a light-weight solution for reference resolution (Section 5.5). In addition, we generalize the grammar supported by NaLIX for followup queries and extend the user interaction mechanism to help users to formulate a followup query that can be understood by the system (Section 5.6). We also develop a technique for automatic query tree discovery to handle "fake" followup queries (Section 5.7). Finally, detailed descriptions on the generation of feedback messages in NaLIX can be found in an Electronic Appendix.

Both this work and its preliminary version [Li et al. 2006] use Schema-Free XQuery [Li et al. 2004, 2007c] as the target language for natural language translation. As discussed in Section 2, Schema-Free XQuery helps to relieve us from the burden of having to map a natural language query to the precise underlying schema. As a result, we were able to focus on issues that are unique to natural language query translation, such as nesting and grouping determination.

7.1 Natural Language Interface to Databases

In the information retrieval field, research efforts have long been made on natural language interfaces that take keyword search query as the target language [Chu-carroll et al. 2002; Delden and Gomez 2004]. In recent years, keyword search interfaces to databases have begun to receive increasing attention [Hulgeri et al. 2001; Cohen et al. 2003; Guo et al. 2003; Hristidis et al. 2003; Li et al. 2004, 2007c], and have been considered a first step toward 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 the information retrieval field in that it supports a richer query mechanism that allows 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 1980s [Androutsopoulos et al. 1995]. 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 [Woods et al. 1972], 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 [Androutsopoulos et al. 1995]. 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 Stallard [1986], our system does not rely on extensive domain-specific knowledge.

The idea of interactive NLIDB was discussed in some early NLIDB literature [Küpper et al. 1993; Androutsopoulos et al. 1995]. The majority of these focused 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 [Kapetanios and Groenewoud 2002; Trigoni 2002]. These depend on domain-specific knowledge. Also, they assist the construction of structured queries rather than natural language queries.

Meng and Chu [1999], Tang and Mooney [2001], and Popescu et al. [2003, 2004] depicted a few notable recent work on NLIDB. Tang and Mooney [2001] presented a learning approach as a combination of learning methods. We view the learning approach 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. Meng and Chu [1999] described an NLIDB based on a query formulator. 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 the query formulator. No experimental evaluation on the effectiveness of the system has been reported. PRECISE [Popescu et al. 2003] is an NLIDB that translates *semantically tractable* NL questions into corresponding SQL queries. While PRECISE extensively depends on database schema for query mapping, NaLIX does not rely on the availability of a schema for query translation. In addition, PRECISE 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 PRECISE with promising quality.

7.2 Dependency Parser

In NaLIX, we obtain the semantic relationships between words via a dependency parser. Recent work in question answering [Attardi et al. 2001; Gao et al. 2004; Cui et al. 2005] has pointed out the value of utilizing the dependency relation between words in English sentences to improve the precision of question answering. Such dependency relations are obtained either from dependency parsers such as MINIPAR [Attardi et al. 2001; Cui et al. 2005] or through statistic training [Gao et al. 2004]. These works all focus on full text retrieval, and thus cannot directly apply to XML databases. Nevertheless, they inspire us to use a dependency parser to obtain semantic relationships between words, as we have done in NaLIX.

7.3 Support for Iterative Database Search

Previous studies have shown that information seeking often involves multiple iterations; thus allowing users to ask followup questions is a highly desired feature in computer-based information-giving systems [Olson et al. 1985; Lauer et al. 1992; Moore 1995]. However, while support for iterative search is common in information retrieval systems [Salton 1971; Saracevic 1997], database systems are typically designed to support one-step search with a few notable exceptions [Guida and Tasso 1983; Carbonell 1983; Carey et al. 1996; McHugh et al. 1997; Goldman and Widom 1998; Egnor and Lord 2000; Sinha and Karger 2005; Ioannidis and Viglas 2006]. Most of such systems, such as PESTO [Carey et al. 1996], DataGuide [McHugh et al. 1997], and Magnet [Sinha and Karger 2005], support followup queries by allowing results browsing as query refinement. Although search in some of the systems (e.g., DataGuide) may start with keyword search or vague query expressions, when handling followup queries they are all essentially visual query systems. As such, they all suffer from the following drawbacks of visual query systems. First, expressing complex queries in such systems, despite the actual visual representation, usually requires multiple steps. In contrast, a user familiar with NaLIX can pose a query in a single step, while a casual user may need two to three iterations to do so, as shown in our user study. Second, query formulation still requires users to understand the semantics of the performed operations. This requirement is nontrivial for casual users—complex concepts such as join, grouping are often found to be ambiguous and cumbersome, leading to difficulties in both query construction and query refinement. Meanwhile, such complex concepts can easily be expressed and understood in natural language. Furthermore, little effort has been made by these systems to organize queries in a meaningful structure, often resulting in the uselessness of query history, whereas a carefully designed iterative search model is used in NaLIX to maintain previous search queries in a meaningful context.

Recent work on conversation querying [Ioannidis and Viglas 2006] accepted incomplete SQL queries as input and allows a user to interact with a database system in a conversational fashion. Besides using different techniques for different types of query input (English sentences vs. SQL), the key differences between our system and that of Ioannidis and Viglas [2006] are the following. First of all, in Ioannidis and Viglas [2006], only the last completed query or last issued query may be modified by a followup query, while in our system, a user may issue a followup query to any arbitrary existing query. In addition, Ioannidis and Viglas [2006] assumed that every followup request performs at most one alternation, while our system does not hold such an assumption.

A few NLIDB systems, including IR-NLI [Guida and Tasso 1983] and XCALIBUR [Carbonell 1983], also support followup queries. However, these systems all attempt to build dialog systems that can reason about the goals and beliefs of the user with limited success. Unlike such systems, we do not attempt to provide a system with superior natural language understanding. Instead, we carefully design interactive feedback mechanism to get the user to formulate system-understandable queries.

7.4 Communication Models

Our model for iterative search is inspired by works in dynamic text messages systems including chat room and discussion forums [Hegngi 1998; Vronay et al. 1999; Smith et al. 2000; Farnham et al. 2000; Sheard et al. 2003]. Hegngi [1998] and Sheard et al. [2003] presented experimental evidence on the advantage of the structured communication model used by discussion forums over the conventional free message flow model used in chat rooms. To overcome the shortcomings of a conventional chat room interface, several interface alternatives for chat have been proposed in recent years [Vronay et al. 1999; Erickson et al. 1999; Viégas and Donath 1999; Smith et al. 2000]. The system most related to our model is Threaded Text by Smith et al. [2000]. This chat environment is based on a notion called *conversation tree*, where the basic turn-taking structure of human conversation is kept for easy comprehension of chat history. Usability study results presented by Smith et al. [2000] and Farnham et al. [2000] suggest that synchronous communications such as text chats can indeed benefit from such structured organization of the messages. Since iterative database search in our system can also be viewed as chat between a user and the system, it is reasonable to believe that users of our system can also benefit from the structured organization of queries in the form of query trees.

7.5 Automatic Topic Discovery

Khan et al. [2002], Kim et al. [2005], and Shen et al. [2006] proposed different automatic topic discovering techniques for text messages: text messages are segmented into threads (similar to a query tree) to facilitate navigation of the messages by keeping messages in a meaningful context. The segmentations are created either by identifying start/end messages based on linguistic features or using clustering algorithms. While these works share a goal similar to that of our extension proposed in Section 5.7, they all analyze text messages in postprocess fashion. In contrast, our system identifies a root query on-the-fly. In addition, we identify a followup query as a new root query based on its semantic relationship with its prior queries instead of its linguistic features.

7.6 Reference Resolution

Reference resolution is an important problem in several fields in natural language processing, such as information extraction, question answering, and summarization. Not surprisingly, many techniques have been proposed to address the problems. These techniques largely fall into the following three categories: (i) machine learning approaches [Aone and Bennett 1995; McCarthy and Lehnert 1995; Ge et al. 1998; Soon et al. 2001; Ng and Cardie 2002]; (ii) knowledge-poor approaches, which depend on simple salience-based rules instead of linguistic or domain knowledge [Mitkov 1998]; and (iii) knowledge-rich approaches, which extensively rely on linguistic and/or domain knowledge [Hobbs 1978; Grosz et al. 1986; Delmonte 1990; Delmonte and Bianchi 1991; Lappin and Leass 1994; Hardt 1996, 2004]. Our approach discussed in Section 5.5 mainly depends on gender and number agreement to decide candidate antecedents and is essentially knowledge-poor. But unlike pure knowledge-poor systems, we also take advantage of knowledge readily available in the system by considering whether a candidate is included in a RETURN clause or is a context center. Since such knowledge is only meaningful for database queries, our technique does not intend to be applicable as a general reference resolution method.

8. 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 article 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 addition, iterative search is supported in the form of followup queries in query history. Along with a query template mechanism, followup queries enable users to incrementally focus their search on the objects of interest with previous search queries preserved in context for reuse should they decide to take a different search direction. In the future, we plan to add support for disjunction, multisentence queries, complex negation, and composite result construction. Our current system is oriented at structured XML databases: we intend to integrate support for phrase matching by incorporating full-text techniques in XQuery such as TeXQuery [Amer-Yahia et al. 2004], 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 article to lead to a whole new generation of query interfaces for databases.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful and constructive comments and suggestions. We are also grateful to Mary Fernández for her comments on a preliminary version of this work. This work has also benefitted from discussions with George Furnas and Dragomir Radev.

REFERENCES

ALIAS-I. 2006. Available online at `http://www.alias-i.com/lingpipe/`.

AMER-YAHIA, S., BOTEV, C., AND SHANMUGASUNDARAM, J. 2004. TeXQuery: A full-text search extension to XQuery. In *Proceedings of the International World Wide Web Conference*. 583–594.

ANDROUTSOPOULOS, I., RITCHIE, G. D., AND THANISCH, P. 1995. Natural language interfaces to databases—an introduction. *J. Lang. Eng. 1*, 1, 29–81.

AONE, C. AND BENNETT, S. 1995. Evaluating automated and manual acquisition of anaphora resolution strategies. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*. Morgan Kaufmann, San Francisco, CA, 122–129.

ATTARDI, G., FORMICA, F., SIMI, M., AND TOMMASI, A. 2001. PiQASso: Pisa question answering system. In *Proceedings of the Text REtrieval Conference*. National Institute of Standards and Technology (Gaithersburg, MD). 633–641.

BATES, M. J. 1989. The design of browsing and berrypicking techniques for the on-line search interface. *Online Rev. 13*, 5, 407–431.

CARBONELL, J. G. 1983. Discourse pragmatics and ellipsis resolution in task-oriented natural language interfaces. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*. 164–168.

CAREY, M. J., HAAS, L. M., MAGANTY, V., AND WILLIAMS, J. H. 1996. PESTO: An integrated query/browser for object databases. In *Proceedings of the International Conference on Very Large Data Bases*. 203–214.

CHU-CARROLL, J., PRAGER, J., RAVIN, Y., AND CESAR, C. 2002. A hybrid approach to natural language Web search. In *Proceedings of the Conference on Empirical Methods on Natural Language Processing*. 180–187.

COHEN, S., MAMOU, J., KANZA, Y., AND SAGIV, Y. 2003. XSEarch: A semantic search engine for XML. In *Proceedings of the International Conference on Very Large Data Bases*. 45–56.

CUI, H., SUN, R., LI, K., KAN, M.-Y., AND CHUA, T.-S. 2005. Question answering passage retrieval using dependency relations. In *Proceedings of the ACM International Conference on Information Retrieval*. 400–407.

CUNNINGHAM, H., MAYNARD, D., BONTCHEVA, K., AND TABLAN, V. 2002. GATE: A framework and graphical development environment for robust NLP tools and applications. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*. 168–175.

DELDEN, S. V. AND GOMEZ, F. 2004. Retrieving NASA problem reports: A case study in natural language information retrieval. *Data Knowl. Eng. 48*, 2, 231–246.

DELMONTE, R. 1990. Semantic parsing with an LFG-based lexicon and conceptual representations. *Comput. Human. 5*, 6, 461–488.

DELMONTE, R. AND BIANCHI, D. 1991. Binding pronominals with an LFG parser. In *Proceedings of the ACL Workshop on Parsing Technologies*. 59–72.

Dimitrov, M., Bontcheva, K., Cunningha, H., and Maynard, D. 2002. A light-weight approach to coreference resolution for named entities in text. In *Proceedings of the Discourse Anaphora and Anaphor Resolution Colloquium*.

Egnor, D. and Lord, R. 2000. Structured information retrieval using XML. In *Proceedings of the ACM SIGIR Workshop on XML and Information Retrieval*. Published on the World Wide Web at `http://www.haifa.il.ibm.com/sigir00-xml/final-papers/Egnor/`.

Erickson, T., Smith, D. N., Kellogg, W. A., Laff, M., Richards, J. T., and Bradner, E. 1999. Socially translucent systems: Social proxies, persistent conversation, and the design of "babble." In *Proceedings of the ACM International Conference on Human Factors in Computing Systems*. 72–79.

Farnham, S., Chesley, H. R., McGhee, D. E., Kawal, R., and Landau, J. 2000. Structured online interactions: improving the decision-making of small discussion groups. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work.* 299–308.

Gao, J., Nie, J.-Y., Wu, G., and Cao, G. 2004. Dependence language model for information retrieval. In *Proceedings of the ACM International Conference on Information Retrieval*. 170–177.

Ge, N., Hale, J., and Charniak, E. 1998. A statistical approach to anaphora resolution. In *Proceedings of the Workshop on Very Large Corpora*. 161–170.

Goldman, R. and Widom, J. 1998. Interactive query and search in semistructured databases. In *Proceedings of the the International Workshop on Web and Databases* (WebDB). 52–62.

Grosz, B., Jones, K. S., and Webber, B. L., Eds. 1986. *Readings in Natural Language Processing*. Morgan Kaufmann Publishers, San Fransisco, CA.

Grosz, B., Joshi, A., and Weinstein, S. 1995. Centering: A framework for modelling the local coherence of discourse. *Computat. Ling. 2*, 21, 203–225.

Guida, G. and Tasso, C. 1983. IR-NLI: An expert natural language interface to online data bases. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*. 31–38.

Guo, L., Shao, F., Botev, C., and Shanmugasundaram, J. 2003. XRANK: Ranked keyword search over XML documents. In *Proceedings of the ACM International Conference on Management of Data*. 16–27.

Hardt, D. 1996. Centering in dynamic semantics. In *Proceedings of the International Conference on Computational Linguistics*. 519–524.

Hardt, D. 2004. Dynamic centering. In *Procceedings of the ACL Reference Resolution Workshop*. 55–62.

Hegngi, Y. N. 1998. Changing roles, changing technologies: The design, development, implementation, and evaluation of a media technology and diversity on-line course. In *Proceedings of the American Educational Research Association Annual Meeting*. 1–30.

Hobbs, J. R. 1978. Resolving pronoun references. *Lingua 44*, 311–338.

Hristidis, V., Papakonstantinou, Y., and Balmin, A. 2003. Keyword proximity search on XML graphs. In *Proceedings of the IEEE International Conference on Data Engineering*. 367–378.

Hulgeri, A., Bhalotia, G., Nakhe, C., Chakrabarti, S., and Sudarshan, S. 2001. Keyword search in databases. *IEEE Data Eng. Bull. 24*, 22–32.

Ioannidis, Y. E. and Viglas, S. D. 2006. Conversational querying. *Inform. Syst. 31*, 1, 33–56.

Jagadish, H. V., Al-Khalifa, S., Chapman, A., Lakshmanan, L. V., Nierman, A., Paparizos, S., Patel, J. M., Srivastava, D., Nuwee Wiwatwattana, Y. W., and Yu, C. 2002. TIMBER: A native XML database. *Int. J. Very Large Data Bases 11*, 4, 274–291.

Kapetanios, E. and Groenewoud, P. 2002. Query construction through meaningful suggestions of terms. In *Proceedings of the International Conference on Flexible Query Answering Systems*. 226–239.

Khan, F. M., Fisher, T. A., Shuler, L., Wu, T., and Pottenger, W. M. 2002. Mining chatroom conversations for social and semantic interactions. Tech. Rep. LU-CSE-02-011, Lehigh University, Bethlehem, PA.

Kim, J. W., Candan, K. S., and Dderler, M. E. 2005. Topic segmentation of message hierarchies for indexing and navigation support. In *Proceedings of the International World Wide Web Conference*. 322–331.

Küpper, D., Storbel, M., and Rösner, D. 1993. NAUDA: A cooperative natural language interface to relational databases. *SIGMOD Rec. 22*, 2, 529–533.

Lappin, S. and Leass, H. 1994. An algorithm for pronominal anaphora resolution. *Comput. Ling. 20*, 4, 535–561.

Lauer, T., Peacock, E., and Graesser, A. 1992. *Questions and Information Systems*. Lawrence Erlbaum Associates, Mahwah, NJ.

Li, Y., Chaudhuri, I., Yang, H., Singh, S., and Jagadish, H. V. 2007a. Danalix: A domain-adaptive natural language interface for querying xml. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

Li, Y., Chaudhuri, I., Yang, H., Singh, S., and Jagadish, H. V. 2007b. Enabling domain-awareness for a generic natural language interface. In *Proceedings of the 22nd Conference on Artificial Intelligence*.

Li, Y., Yang, H., and Jagadish, H. V. 2005. NaLIX: An interactive natural language interface for querying XML. In *Proceedings of the ACM International Conference on Management of Data*. 900–902.

Li, Y., Yang, H., and Jagadish, H. V. 2006. Constructing a generic natural language interface for an XML database. In *Proceedings of the International Conference on Extending Database Technology*. 737–754.

Li, Y., Yu, C., and Jagadish, H. V. 2004. Schema-Free XQuery. In *Proceedings of the International Conference on Very Large Data Bases*. 72–83.

Li, Y., Yu, C., and Jagadish, H. V. 2007c. Enabling Schema-Free XQuery with Meaningful Query Focus. *Int. J. Very Large Databases*. To appear. DOI = 10.1007/s00778-006-0003-4. Available online at `http://www.spingerlink.com/content/7th8518314515x74/`.

Lin, D. 1998. Dependency-based evaluation of MINIPAR. In *Proceedings of the Workshop on the Evaluation of Parsing Systems* (Granada, Spain).

McCarthy, J. F. and Lehnert, W. G. 1995. Using decision trees for coreference resolution. In *Proceedings of the International Joint Conferences on Artificial Intelligence*. 1050–1055.

McHugh, J., Abiteboul, S., Goldman, R., Quass, D., and Widom, J. 1997. Lore: A database management system for semistructured data. *SIGMOD Rec. 26*, 3 (Sept.), 54–66.

Mel'čuk, I. A. 1979. *Studies in Dependency Syntax*. Karoma Publishers, Ann Arbor, MI.

Meng, F. and Chu, W. 1999. Database query formation from natural language using semantic modeling and statistical keyword meaning disambiguation. Tech. Rep. CSD-TR 990003, University of California, Los Angels, Los Angels, CA.

Miller, G. A., Beckwith, R., Fellbaum, C., Gross, D., and Miller, K. 1990. Five papers on WordNet. *Int. J. Lexicol. 3*, 4, Available online at `http://www.cogsci.princeton.edu/~wn/`.

Mitkov, R. 1998. Robust anaphora resolution with limited knowledge. In *Proceedings of the International Conference on Computational Linguistics*.

Moore, J. D. 1995. *Participating in Explanatory Dialogues Interpreting and Responding to Questions in Context*. MIT Press, Cambridge, MA.

Ng, V. and Cardie, C. 2002. Identifying anaphoric and non-anaphoric noun phrases to improve coreference resolution. In *Proceedings of the International Conference on Computational Linguistics*.

Olson, G. M., Duffy, S. A., and Mack, R. L. 1985. *The Psychology of Questions*. Lawrence Erlbaum Associates, Mahwah, NJ. Chapter 7, 219–227.

Popescu, A.-M., Armanasu, A., Etzioni, O., Ko, D., and Yates, A. 2004. Modern natural language interfaces to databases. In *Proceedings of the International Conference on Computational Linguistics*. 141–147.

Popescu, A.-M., Etzioni, O., and Kautz, H. 2003. Towards a theory of natural language interfaces to databases. In *Proceedings of the International Conference on Intelligent User Interfaces*. 149–157.

Quirk, R., Greenbaum, S., Leech, G., and Svartvik, J. 1985. *A Comprehensive Grammar of the English Language*. Longman, London, U.K.

Remde, J. R., Gomez, L. M., and Landauer, T. K. 1987. Superbook: An automatic tool for information exploration—hypertext? In *Proceedings of the ACM Conference on Hypertext and Hypermedia*. 175–188.

Salton, G. 1971. *The SMART Retrieval System*. Prentice Hall, Upper Saddle River, NJ.

SARACEVIC, T. 1997. The stratified model of information retrieval interaction: Extension and applications. In *Proceedings of the American Society for Information Science & Technology Annual Meeting*.

SCHMIDT, A., KERSTEN, M., AND WINDHOUWER, M. 2001. Querying XML documents made easy: Nearest concept queries. In *Proceedings of the IEEE International Conference on Data Engineering*, 321–329.

SHAW, W. M., BURGIN, R., AND HOWELL, P. 1997. Performance standards and evaluations in IR test collections: Vector-space and other retrieval models. *Inform. Process. Manage. 33*, 1, 15–36.

SHEARD, J., MILLER, J., AND RAMAKRISHNAN, S. 2003. Web-based discussion forums: The staff perspective. In *Proceedings of the Annual conference on Innovation and Technology in Computer Science Education*. 158–162.

SHEN, D., YANG, Q., SUN, J.-T., AND CHEN, Z. 2006. Thread detection in dynamic text message streams. In *Proceedings of the ACM International Conference on Information Retrieval*. 35–42.

SINHA, V. AND KARGER, D. 2005. Magnet: Supporting navigation in semistructured data environments. In *Proceedings of the ACM International Conference on Management of Data*. 97–106.

SLEATOR, D. AND TEMPERLEY, D. 1993. Parsing English with a link grammar. In *Proceedings of the International Workshop on Parsing Technologies*.

SMITH, M., CADIZ, J. J., AND BURKHALTER, B. 2000. Conversation trees and threaded chats. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*. 97–105.

SOON, W., NG, H., AND LIM, D. 2001. A machine learning approach to coreference resolution of noun phrases. *Computat. Ling. 27*, 4, 521–544.

STALLARD, D. 1986. A terminological transformation for natural language question-answering systems. In *Proceedings of the Australasian Natural Language Processing Workshop*. 241–246.

TANG, L. R. AND MOONEY, R. J. 2001. Using multiple clause constructors in inductive logic programming for semantic parsing. In *Proceedings of the European Conference on Machine Learning*. 466–477.

TRIGONI, A. 2002. Interactive query formulation in semistructured databases. In *Proceedings of the International Conference on Flexible Query Answering Systems*. 356–369.

VIÉGAS, F. B. AND DONATH, J. S. 1999. Chat circles. In *Proceedings of the ACM International Conference on Human Factors in Computing Systems*. 9–16.

VRONAY, D., SMITH, M., AND DRUCKER, S. 1999. Alternative interfaces for chat. In *Proceedings of the ACM Symposium on User Interface Software and Technology*. 19–26.

WOODS, W., KAPLAN, R. M., AND NASH-WEBBER, B. 1972. *The Lunar Sciences Natural Language Information System: Final Report*. Bolt Beranek and Newman Inc., Cambridge, MA.

WORLD WIDE WEB CONSORTIUM. 2003. XML Query Use Cases. World Wide Web Consortium Working Draft. Available online at `http://www.w3.org/TR/xquery-use-cases/`.

WORLD WIDE WEB CONSORTIUM. 2004. Extensible Markup Language (XML) 1.0 (Third Edition). World Wide Web Consortium Recommendation. Available online at `http://www.w3.org/TR/REC-xml/`.

Received November 2006; revised May 2007; accepted August 2007