Implementing A Linguistic Query Language for Historic Texts

Lukas C. Faulstich and Ulf Leser and Thorsten Vitt

Institut für Informatik
{faulstic,leser,vitt}@informatik.hu-berlin.de
Humboldt-Universität zu Berlin
Unter den Linden 6, D-10099 Berlin

Abstract. We describe design and implementation of the linguistic query language DDDquery. This language aims at querying a large linguistic database storing a corpus of richly annotated historic German texts. We use a graph-based data model that supports multiple independent annotation layers on a shared text layer as well as alignments of text layers representing the same text or related texts (e.g., translations). The corpus is stored in an object-relational database system with a text retrieval extension.

DDDquery is based on XPath to leverage the familiarity of many users with this language. It is translated to SQL in a two phase compilation with first order logic as an intermediate language. This approach effectively decouples the query language from the schema of the underlying corpus.

We provide an overview of DDDquery, the underlying ODAG data model, its implementation as relational schema, the predicates of the intermediate language, and describe both phases of the translation process.

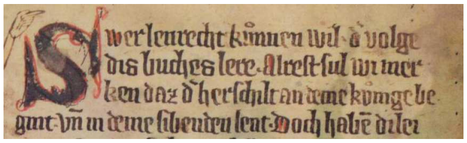
1 Introduction

The project DDD[1] is a large interdisciplinary project of linguists of historical German, corpus linguists, computational linguists, and computer scientists for creating a *diachronic corpus* of German, i.e., a collection of German texts ranging from the 8th century to modern German carefully selected to cater linguistic research interests. Most texts in the DDD corpus will be richly annotated, i.e., words will be annotated with morphological, lexical, and grammatical information; sentences will be annotated with their syntactic structure; and whole texts will be annotated with respect to the structure of their content as well as with bibliographic and other meta-data [1].

In the DDD project, we are developing methods to store and manage large collections of richly annotated historic texts such as the Sachsenspiegel[2] in an RDBMS [2]. The project is faced with non-tree-shaped annotation graphs and multiple annotation hierarchies with conflicting structure that cannot be represented naturally in XML. For

[1] www.deutschdiachrondigital.de

[2] The "Sachsenspiegel" is the earliest code of common law written in German. The Heidelberg manuscript, a Middle High German version of the "Sachsenspiegel", is available at http://digi.ub.uni-heidelberg.de/cpg164; a detail is shown in Fig. 1



Swer lenrecht ku°nnen wil. d' volge
dis buches lere. Alreſt ſul wi mer
ken daz d' herſchilt an deme koninge be
gint. vn̄ in deme ſibenden leat. Doch habē di lei

Fig. 1. Detail from page 1r of the “Heidelberger Sachsenspiegel”.

instance, the logical organization of a text in sections, paragraphs, sentences, and words often conflicts with the structure of its physical source (e.g., a manuscript) in pages, lines, and whitespace-separated groups of characters: sentences may cross several lines, logical words may be hyphenated etc. Historical linguists use several parallel text layers (e.g., a so-called *diplomatic* version close to the original text witness, a more readable *normalized* version, a word-by-word translation, alternative versions from different text witnesses etc.) which need to be carefully aligned with each other (cf. Fig. 2). Searching within text layers, in different annotation layers and across alignments in combination with hierarchical and spatial relationships (e.g., precedence, inclusion, intersection of text spans) poses further challenging requirements to the query language.

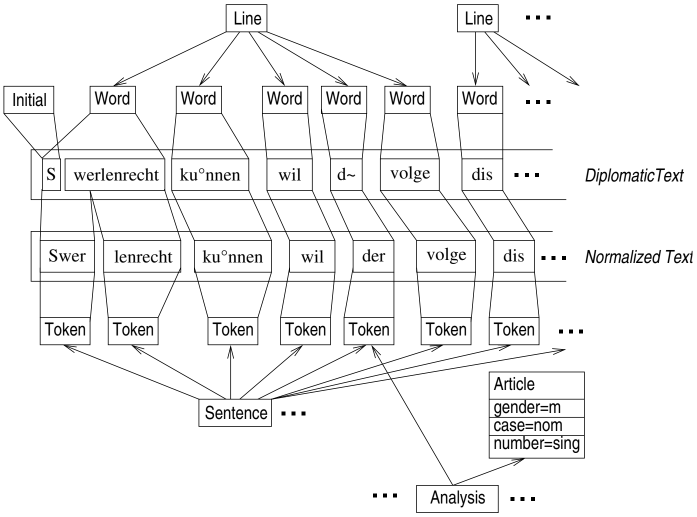


Fig. 2. Exemplary annotation of the Sachsenspiegel (detail; cf. Fig. 1).

While in principle XML together with a reference mechanism such as XPointer can be used to encode richly annotated corpora by means of so-called stand-off annotation, we believe (with others [3,4]) that specialized data models are a more natural and promising way to cope with the requirements of such corpora. Hence we have introduced [5] the graph-based ODAG data model presented in Sec. 2 that extends the XML data model. The particular requirements of linguists for querying corpora have led to the development of various linguistic query tools and specialized query languages such as CQP [6] and TigerSearch [7]. Recently, there have been proposals for XML-based linguistic query languages such as LPath [8]. While LPath is designed for querying tree banks encoded in XML, we propose the query language DDDquery for querying richly annotated corpora represented in our ODAG model. It goes beyond LPath by supporting queries on text spans, on multiple annotation layers, and across aligned texts.

1.1 Requirements and Design Decisions

Our query language should build on an established and popular standard to leverage the familiarity of users with this base language. We have decided to base the *syntax* of our language on XPath due to its popularity, its simplicity (compared to XQuery) and the similarity of our data model to XML. We need to extend XPath with linguistic query operations such as projection through alignments and selection of text spans by content (full-text retrieval) or according to spatial relationships (textual order).

As search results neither lists of whole documents (like in Information Retrieval) nor linear sequences of nodes (like in XPath) are sufficient. A query needs to specify several targets (e.g., a sentence together with a verb and noun phrase within this sentence) the matches of which must be shown in their textual context together with selected annotations. The result of a DDDquery query is a subgraph of the corpus where matches of particular targets are specially tagged to facilitate highlighting by the presentation layer that is in charge of formatting the results. Hence construction of arbitrary XML documents (as supported by XQuery) is deliberately not offered by DDDquery.

While the DDD corpus will be stored in a object-relational database system, the language should be independent from the implementation of this storage layer. Hence we choose first-order logic over a fixed set of primitive predicates as an intermediary language that abstracts from the underlying database schema.

With the corpus still in its planning stage, user requirements cannot be fully predicted. Hence we need an easily extensible language. Using the JavaCC and JJTree tools facilitates syntactical extensions. New primitive predicates can easily be defined in the logic-based intermediate layer in terms of SQL templates without changing our logic-to-SQL compiler LoToS. Moreover, changes of the underlying relational schema can be compensated to some extent by adapting the SQL templates.

1.2 Structure of the Paper

In the next section we discuss the underlying ODAG data model and how it is implemented as relational schema. Then, in Sec. 3 we give an overview of DDDquery. The intermediate language is presented in Sec. 4. Then we discuss the two translation phases in Sec. 5 and Sec. 6 before we conclude the paper with Sec. 7.

2 The ODAG Linguistic Data Model

The principle elements of the ODAG linguistic data model are depicted in Fig. 3: Text layers are represented by a text ID and the actual text content. Spans are continuous intervals of texts, represented by their left and right border and by an optional score (which may result from an approximate full text search). Elements, characterized by a name and further described by a set of Attributes, may refer to a span. An element may have an ordered sequence of children and one or more parents, thus forming a DAG. Cycles however are forbidden.

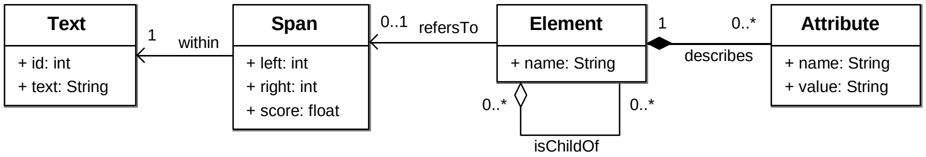


Fig. 3. The ODAG data model in form of an UML class diagram.

The relational schema presented here is a straight-forward implementation of this data model. It is based on [9]. Text layers are stored in a table

`text(id, content)`

where `content` is a string (CLOB) storing the text of the text layer identified by attribute `id`. ODAG elements are stored in table

`element(id, name, tid, left, right, score)`

where `tid` refers to the identifier of a text layer. If `tid` is not null, then the tuple `(tid,left,right,score)` refers to a text span associated with the element.

The children of each element are stored in table

`child(parent, pos, child)`

where `parent` and `child` reference `element.id` and `pos` indicates the position of a child within its siblings.

Since the ODAG data model generalizes XML to acyclic directed graphs, our storage concept is based on a shredded interval-based storage scheme similar to [10]. In this model, each document node is stored together with its so-called pre- and post-order ranks. These ranks result from traversing the document tree depth-first and numbering each node before (pre-order rank) and after (post-order rank) visiting its children. This representation allows queries for the XPath axes to be translated into simple conditions on rank-intervals. This approach has been generalized in [11] to support graph-based data models such as ODAG. An ODAG element that can be reached on different paths will be visited multiple times during a complete traversal of the ODAG. Hence for each visit a different rank is attached to the element:

`rank(element, pre, post)`

Attribute `pre` stores a pre-order rank of the element referenced by attribute `element` and attribute `post` stores the corresponding post-order rank.

The attributes of ODAG elements are stored in table

`attribute(element, name, value)`

An attribute is uniquely identified by the `id` of the element it describes (referred to by attribute `element`) and its `name`.

3 Overview of the Query Language DDDquery

This section gives a brief overview of DDDquery. For a detailed presentation see [14]. DDDquery extends XPath to fulfill the linguists' requirements and to handle the data model outlined in Sec. 2.

As in XPath, DDDquery's fundamental language element is a *path expression* composed of *location steps*. Each step consists at least of an *axis* and a *node test* and may optionally contain *predicates* (which further constrain the set of matched nodes) and *variables* (explained below). A path expression in a query matches a set of paths in the corpus graph such that the steps match graph nodes and the axes describe the relations between the respective nodes.

3.1 Complex Query Features

Linguists' requirements [12] for corpus query languages include regular expressions on path components and correlation of subqueries. For instance, sample query Q_1 from [8] "Noun phrases *NP* that immediately follow a verb *V*" (within the same Sentence *S*) can be expressed in DDDquery using a shared variable `$NP` by

```
    //S$S//V$V//immediately-following::NP$NP & $s//NP$NP
```

While this query can be expressed as well in LPath (but not in XPath), queries involving more complex correlations can not. Further features that go beyond LPath such as alignments and multiple layers are supported using dedicated axes and elements.

3.2 Complex Search Results

Unlike XPath, where the path's result is simply a sequence of nodes "pointed at" by the path expression, DDDquery needs to offer more complex query results, e. g. context information. Hence for each step the matching nodes can be selected for output by binding the step to a variable. The variable name will be used to annotate the respective nodes in the result, such that users and front-ends can recognize the mapping between result nodes and query steps. The subgraph induced by the nodes selected for output will be the result of the query. For example, `//S$s//VP//NP$np` selects all NP elements which are descendants of VP elements which are descendents of S elements, but only outputs the NP and S elements, marking them with "s" and "np", respectively.

The purpose of DDDquery is to provide structured results containing all necessary information needed for their presentation in a Web interface, but not to prepare documents in a presentation format such as HTML. Hence it does not provide constructs for assembling arbitrary XML documents but returns the results in a fixed XML format, which can then be post-processed using, for instance, XSLT.

3.3 Spans and Full Text Search

Spans (i. e. continuous intervals of text) are first class objects of the corpus data model. They may be initially obtained either by following an association with an element node or by a full text search in the document. We cope with spans by introducing special node tests. For instance, DDDquery provides a node test `exact-match('Siegfried')` which matches all text spans consisting of the exact string “Siegfried”. A similar test is defined for regular expression matching, and the language is open for extension with additional tests like a fuzzy text search.

To describe relations between spans there are respective axes like, e. g., `contained`. In particular, the semantics of the horizontal XPath axes like `following` and `preceding` have been adapted to refer to relations between spans. To allow navigation from an element to the associated span and vice versa, we provide special axes. E. g., the path fragment

```
exact-match("XPath") / containing-element::sentence /
  contained::exact-match("grammar")
```

navigates from a text span with the content *XPath* to a text span *grammar* which must be contained in the same sentence.

3.4 Syntactic Sugar

Like XPath, DDDquery provides a simple, but rather verbose normative syntax able to express all language features plus a set of abbreviations for common constructs. In particular there are, like in LPath [8], shortcut symbols for axis steps using many common axes (so `a/following::b` may be abbreviated as `a --> b`).

4 Intermediate Language

Queries in DDDquery are translated to an intermediate language called DDDlog that is based on first-order logic with a fixed set of predicates. In DDDlog a query q is defined as a horn clause $H \leftarrow F$. The head $H = h(X_1,\dots,X_m)$ defines the variables $X_1,\dots,X_m$ as parameters of q. F is a Boolean formula defined recursively as $F ::= F_1 \vee F_2 | F_1 \wedge F_2 | \neg(F_1) | p(t_1,\dots,t_n) | p(t_1,\dots,t_n)^+$. h and p are predicate symbols. The call parameters t_j are either constants or variables. The transitive closure operator $()^+$ provides a limited form of recursion that can be handled by current DBMS. A call $p(t_1,\dots,t_n)^+$ is equivalent to a call $r(t_1,\dots,t_n)$ to a recursive predicate r/n defined as

$$r(X_1,\dots,X_n) \equiv p(X_1,X_2,\dots,X_{n-1},Y) \wedge (Y = X_n \vee r(Y,X_2,\dots,X_{n-1},X_n))$$

A query result is a substitution σ for all free variables in q such that the result $F\sigma$, i.e., applying σ to F, is true.

A predicate is either a *macro* or a *primitive*. Macros are defined by a set of non-recursive horn clauses, i.e., a macro defines an intensional database predicate. Macros are expanded before query translation. Primitives are defined in terms of templates that can be instantiated to SQL code. Templates are discussed in Sec. 6.

4.1 Representation of Corpus Nodes

The semantics of DDDquery is defined in terms of *(corpus) nodes*. Similar to document nodes in XML there are different types of corpus nodes.

A corpus node is either a span, an instance of an element, or an instance of an attribute. An element instance is an element together with a pre- and post-order rank for this element and the optional span associated with this element. For each instance of an element in combination with one of the element's attributes there exists an attribute instance. Since our FO-to-SQL compiler does not support polymorphic types, we represent all nodes by the same tuple type

$$(\mathit{EltId}, \mathit{Name}, \mathit{Value}, \mathit{Pre}, \mathit{Post}, \mathit{TextId}, \mathit{Left}, \mathit{Right}, \mathit{Score})$$

For spans only the last four components are not null. For element instances, component *Name* stores the element name while *value* is null. For attribute instances, name and value of the attribute are stored in the respective components. Attribute instances are not associated with spans, hence the last four components are always null.

4.2 Predicates of the Intermediate Language

DDDlog defines a set of predicates for node tests (discussed in Sec. 5.1) and axis steps (presented in Sec. 5.2). They are defined as macros building on other predicates: (i) primitives providing the relational tables presented in Sec. 2 as extensional database predicates, (ii) auxiliary predicates (s. Table 1) for accessing components of nodes, (iii) basic predicates (s. Table 2) defining relationships between nodes, and (iv) primitives providing access to various SQL functions.

Due to space limitations we cannot describe all predicates of the intermediate language but must limit the presentation to some examples.

Predicate	**Definition**
$\mathit{nodeRank}(E,P,Q)$	$E = (I,N,V,P,Q,T,L,R,C)$
$\mathit{span}(E,T,L,R,S)$	$T \neq \mathit{null} \wedge E = (I,N,V,P,Q,T,L,R,S)$
$\mathit{elementId}(E,I)$	$E = (I,N,V,P,Q,T,L,R,C)$

Table 1. Auxiliary Predicates.

Predicate	**Definition**
$\mathit{ancestor}(A,D)$	$\mathit{nodeRank}(A,P_A,Q_A) \wedge \mathit{nodeRank}(D,P_D,Q_D) \wedge P_A < P_D \wedge Q_D < Q_A$
$\mathit{immPrec}(X,Y)$	$\mathit{span}(X,T_X,L_X,R_X,S_X) \wedge \mathit{span}(Y,T_Y,L_Y,R_Y,S_Y) \wedge T_X = T_Y \wedge R_X = L_Y$
$\mathit{alias}(X_1,X_2)$	$X_1 = (I,N,V,P_1,Q_1,T,L,R,C) \wedge X_2 = (I,N,V,P_2,Q_2,T,L,R,C)$

Table 2. Basic Predicates defined as macros.

With the help of the auxiliary predicate $\mathit{nodeRank}/3$ we can define predicate $\mathit{ancestor}/2$ as a relation on nodes. Similarly we can define spatial predicates such as $\mathit{immPrec}/2$ on nodes that have spans on the same text layer. $\mathit{immPrec}(X,Y)$ is satisfied if the right boundary of the span of X coincides with the left boundary of the span of Y.

The predicate $alias/2$ tests whether two nodes are instances of the same element.

The textual content of a node associated with a span is computed by the primitive predicate $content(X,S)$ that is defined by a SQL template since it needs to use the SQL function `SUBSTR()` for computing a substring of a text layer (s. Example 2 in Sec. 6.1).

For full-text retrieval we assume several primitives like $matches(P,S,M)$ that take a string P conforming to a certain syntax (e.g., for regular expressions) and a node S with a span and return all sub-spans M that match P. Variants include predicates for matching a pattern only against the whole span S or against prefixes or suffixes of S. Since SQL and typical full-text retrieval extensions in existing database systems do not support this type of operation well, we may need to support these primitives by appropriate table functions.

5 Translation to DDDlog

The DDDquery-to-DDDlog translation is implemented in JavaCC and JJTree.

5.1 Translation of Node Tests

Node tests are used to generate bindings for node variables. In Table 3 node tests for element and attribute instances are presented that take an (optional) argument N specifying the element/attribute name. Node tests for spans are treated in Sec. 5.3.

Predicate	**Definition**
$elementNode(N,E)$	$element(I,N,T,L,R,C) \wedge rank(I,P,Q) \wedge E = (I,N,null,P,Q,T,L,R,C)$
$attributeNode(N,A)$	$attribute(I,N,V) \wedge rank(I,P,Q) \wedge A = (I,N,V,P,Q,null,null,null,null)$

Table 3. Node tests.

5.2 Translation of Axes

Each axis is implemented by a predicate that defines a node relation (s. Table 4) without necessarily computing node bindings for the target nodes. Note that $ancestorAxis/2$ needs to find also ancestors A not on the path on which the descendent D has been reached. Hence we need first to find all alias nodes D' of D (i.e., all element instances sharing the element with D, but representing different paths to reach this element).

Predicate	**Definition**
$childAxis(P,C)$	$elementId(P,E_P) \wedge elementId(C,E_C) \wedge child(E_P,\_,E_C)$
$parentAxis(C,P)$	$elementId(C,E_C) \wedge elementId(P,E_P) \wedge child(E_P,\_,E_C)$
$descendentAxis(A,D)$	$ancestor(A,D)$
$ancestorAxis(D,A)$	$elementNode(\_,D') \wedge alias(D,D') \wedge ancestor(A,D')$

Table 4. Axis steps.

5.3 Combination of Axis Steps and Node Tests

The primitives for text pattern matching mentioned in Sec. 4 conceptually combine an axis step (e.g., computing all sub-spans of a span) with a node test (e.g., testing whether the text content of a sub-span equals a given string). It would be extremely inefficient to actually generate a large number of spans and then filter those spans satisfying the node test. Hence we translate combinations of axis steps on spans and node tests involving text pattern matching together. For instance, the query fragment

```
contains::exact-match('word')
```

is translated to a call $\mathit{exactMatchSubstring}(\texttt{"word"}, S, M)$. The query fragment

```
following::re-match('be.{1,5}en')
```

is translated to $\mathit{suffixAfter}(S,T) \wedge \mathit{regexpMatchSubstring}(\texttt{"be.\{1,5\}en"}, T, M)$ where $\mathit{suffixAfter}(S,T)$ returns for a span S the suffix T of the whole text layer starting at the right border of S.

In some cases, it is more efficient to bind spans in some other part of the query and just test whether they match the pattern. This results in a different translation that avoids to call a table function for text pattern matching. For instance,

```
contains::exact-match('where')/element::word
```

translates to $\mathit{elementNode}(\_, \texttt{"word"}, W) \wedge \mathit{content}(W, \texttt{"where"})$.

5.4 Example

The sample query `//S$S//V$V//immediately-following::NP$NP & $s//NP$NP` introduced in Sec. 3.1 is translated to the following predicate definition:

$$\begin{aligned} Q_1(S,V,NP) \equiv\ & \mathit{elementNode}(\texttt{'S'},S)\ \wedge \\ & \mathit{ancestor}(S,V)\ \wedge\ \mathit{elementNode}(\texttt{'V'},V)\ \wedge\ \mathit{immPrec}(S,NP) \\ & \mathit{ancestor}(S,NP)\ \wedge\ \mathit{elementNode}(\texttt{'NP'},NP) \end{aligned}$$

6 Translation from DDDlog to SQL

6.1 Templates

A primitive predicate p is defined by one or more templates T, each of which provides an SQL implementation for a certain binding pattern that can be instantiated to an SQL SELECT statement. The FO-to-SQL compiler combines these templates in such a way that for every variable in a query there is a template that binds this variable. Primitives are not necessarily extensional database predicates since their templates may combine data from multiple tables of the underlying database and may contain calls to SQL functions (e.g., for full-text search).

Definition 1 (Template).

A template T for a predicate $p(a_1,\ldots,a_m)$ has the form (A,I,R,σ,τ,w) where

- *$A = \langle a_1,\ldots,a_m\rangle$ is the parameter vector of p.*
- *$I \subseteq \{a_1,\ldots,a_m\}$ is a set of* input parameters *that must be bound externally.*
- *$O = \{a_1,\ldots,a_m\} - I$ is the set of* output parameters *that can be computed by p.*
- *$R = \{r_1,\ldots,r_n\}$ is a finite set of* table aliases

- $\mathscr{E}_{R,I}$ *is the set of SQL expressions over aliases R and free variables I.*
- $\sigma : O \rightarrow \mathscr{E}_{R,I}$, *the* output substitution, *assigns expressions to output parameters.*
- $\tau : R \rightarrow \mathscr{T} \cup \mathscr{Q}_{R,I}$, *the* table assignment, *assigns each table alias an element from the set* $\mathscr{T}$ *of table names or from the set* $\mathscr{Q}_{R,I}$ *of sub-queries over R and I.*
- $w \in \mathscr{E}_{R,I}$ *is an SQL condition that must be satisfied for each solution of p.*

Example 1. The template for predicate $element(E,N,T,L,R,C)$ may be defined as

$$\begin{aligned} T_{element} &= (\langle E,N,T,L,R,C\rangle, \{\mathtt{e}\}, \sigma_{element}, \{\mathtt{e} \mapsto \mathtt{element}\}, true) \\ \sigma_{element} &= \{E \mapsto \mathtt{e.id}, N \mapsto \mathtt{e.name}, \\ &\qquad T \mapsto \mathtt{e.tid}, L \mapsto \mathtt{e.left}, R \mapsto \mathtt{e.right}, C \mapsto \mathtt{e.score}\} \end{aligned}$$

A call to EDB predicate *element*, for instance $element(\mathtt{"e21"},N,T,L,R,C)$, induces a binding β for all (here: zero) input parameters and some output parameters, i.e., $\beta(E) = \mathtt{"e21"}$. The template can be expanded then to the following SQL query which retrieves the name and span for element `"e21"`:

```
SELECT e.name, e.tid, e.left, e.right, e.score
FROM element e
WHERE e.id= "e21"
```

Example 2. The content of a span is the substring of the underlying text layer starting from the left span boundary up to the right span boundary. The predicate $content(T_0,S,X)$ is fulfilled if node X is associated with a span in text layer T_0 whose content is string S. The following template supports the binding pattern where $X = (I,N,V,P,Q,T,L,R,C)$ is given while T_0 and S are requested:

$$\begin{aligned} T_{content} = (&\langle T_0,S,I,N,V,P,Q,T,L,R,C\rangle, \\ &\{T,L,R\}, \\ &\{\mathtt{t}\}, \\ &\{T_0 \mapsto \mathtt{t.id}, S \mapsto \mathtt{SUBSTR}(\mathtt{t.content},L,R-L)\}, \\ &\{\mathtt{t} \mapsto \mathtt{text}\}, \\ &(\mathtt{t.id} = T \ \mathtt{AND}\ \mathtt{T.id} <> \mathtt{null})) \end{aligned}$$

6.2 Translation Algorithm

Although FO-to-SQL translation is standard textbook knowledge, we are not aware of any ready-to-use implementation. Moreover, we wanted to support the transitive closure operator by using the recursive SQL constructs provided now by several DBMS. Hence we developed the **Lo**gic **To** **S**QL compiler LoToS[3] described in [13]. Depending on the underlying database system the transitive closure operator is translated into a `WITH RECURSIVE ... SELECT` statement (e.g., DB2) or using the `SELECT ... CONNECT BY` construct (Oracle).

[3] Available at `http://www.informatik.hu-berlin.de/˜faulstic/projects/DDD/software/LoToS/`

6.3 Result Construction

Executing a DDDquery compiled to SQL produces a result table in which each tuple represents the bindings of the query variables to corpus nodes. Our goal is to generate a result document in which all relevant variable bindings are highlighted. Hence we need a post-processing step in which the union of all variable bindings is computed and sorted by pre-order rank to allow for the construction of a result document tree.

6.4 Example

The DDDlog representation of sample query Q_1 (cf. Sec. 5.4) is translated by LoToS to the SQL query listed in Table 5 that can be executed against a corpus stored using the schema presented in Sec. 2.

```
SELECT
  element1.id,
  element3.id,
  element2.id
FROM
  element element1,
  rank ancestor1,
  rank descendant1,
  rank ancestor2,
  rank descendant2,
  element element2,
  element element3
WHERE element1.name="S"
AND element2.name="V"
AND element3.name="NP"
AND element1.id=ancestor1.element
AND ancestor1.element=ancestor2.element
AND descendant1.element=element2.id
AND descendant2.element=element3.id
AND ancestor1.pre=<descendant1.pre
AND descendant1.pre<ancestor1.post
AND ancestor2.pre=<descendant2.pre
AND descendant2.pre<ancestor2.post
AND (element2.tid IS NOT NULL)
AND (element3.tid IS NOT NULL)
AND element2.tid=element3.tid
AND element2.right=element3.left;
```

Table 5. SQL code for sample query Q_1

7 Conclusion and Future Work

We have given an overview of the linguistic query language DDDquery and its implementation. We are currently integrating the DDDquery parser developed in [14] with the generic FO-to-SQL compiler LoToS [13] into a linguistic query processor. This processor will be tested on a small sample corpus. Thorough performance tests on large corpora need to be undertaken. Until a significant part of the DDD corpus becomes available, we need to use synthetic data or data from existing corpora. Another open point is how to support full-text retrieval on historic texts. The requirements of historic linguists go beyond what is supported in typical full-text indexing solutions: proper Unicode support, regular expression and substring matching, finding all matches within texts rather than all matching texts etc. The two-phase compilation approach taken by DDDquery allows to tune its relational implementation easily without touching the first translation stage and the generic FO-to-SQL compiler.

References

1. Lüdeling, A., Poschenrieder, T., Faulstich, L.C.: DeutschDiachronDigital - Ein diachrones Korpus des Deutschen. In: Jahrbuch für Computerphilologie 6 (2004). Mentis Verlag (2005) 119–136
2. Faulstich, L.C., Leser, U., Lüdeling, A.: Storing and Querying Historical Texts in a Relational Database. Informatik-Bericht 176, Inst. für Informatik, Humboldt-Universität zu Berlin (2005)
3. Carletta, J., Kilgour, J., O'Donnell, T., Evert, S., Voormann, H.: The NITE object model library for handling structured linguistic annotation on multimodal data sets. In: 3rd Workshop on NLP and XML, NLPXML-2003. (2003)
4. Dekhtyar, A., Iacob, I.E.: A framework for management of concurrent XML markup. Data and Knowledge Engineering **52** (2005) 185–208
5. Dipper, S., Faulstich, L.C., Leser, U., Lüdeling, A.: Challenges in Modelling a Richly Annotated Diachronic Corpus of German. In: Workshop on XML-based richly annotated corpora, Lisbon, Portugal (2004)
6. Christ, O.: A modular and flexible architecture for an integrated corpus query system. In: COMPLEX'94, Budapest (1994)
7. Lezius, W.: Ein Suchwerkzeug für syntaktisch annotierte Textkorpora. PhD thesis, Institut für maschinelle Textverarbeitung (IMS), Universität Stuttgart (2002)
8. Bird, S., Chen, Y., Davidson, S., Leea, H., Zheng, Y.: Extending XPath to Support Linguistic Queries. In: Workshop on Programming Language Technologies for XML (PLAN-X). (2005)
9. Vitt, T.: Speicherung linguistischer Korpora in Datenbanken. Studienarbeit, Institut für Informatik, Humboldt Universität zu Berlin (2004)
10. Grust, T., Keulen, M.V., Teubner, J.: Accelerating XPath evaluation in any RDBMS. ACM Transactions on Database Systems **29**(1) (2004) 91–131
11. Trissl, S., Leser, U.: Querying ontologies in relational database systems. In: 2nd Conference on Data Integration in the Life Sciences (DILS05). (2005)
12. Lai, C., Bird, S.: Querying and updating treebanks: A critical survey and requirements analysis. In: Proceedings of the Australasian Language Technology Workshop. (2004)
13. Faulstich, L.C., Leser, U.: Implementing Linguistic Query Languages Using LoToS. Informatik-Bericht 195, Institut für Informatik, Humboldt-Universität zu Berlin (2005)
14. Vitt, T.: DDD*query*: Anfragen an komplexe Korpora. Diplomarbeit, Institut für Informatik, Humboldt-Universität zu Berlin (2005)