Implementing Linguistic Query Languages Using LoToS

Lukas C. Faulstich[1], und Ulf Leser[1]

[1]{faulstic,leser}@informatik.hu-berlin.de
Institut für Informatik, Humboldt-Universität zu Berlin

18th November 2005

Abstract

A linguistic database is a collection of texts where sentences and words are annotated with linguistic information, such as part of speech, morphology, and syntactic sentence structure. While early linguistic databases focused on word annotations, and later also on parse-trees of sentences (so-called *treebanks*), the recent years have seen a growing interest in richly annotated corpora of historic texts that include not only syntactic annotations but further complex annotations, such as alignments between related text layers. This raises the issue of efficiently querying such complex structured linguistic databases.

We present a generic approach for defining domain-specific query languages that we use in developing a query language for richly annotated historic corpora. In our approach, a query language is defined as a set of predicates. A query in form of a logic rule is translated by our LoToS query compiler into a *single*, possibly deeply nested *SQL query*.

In contrast to previous approaches, the annotation structures that can be queried need not be trees but can also form DAGs, or, for a restricted class of recursive queries, arbitrary graphs. To this end, LoToS offers an operator for computing transitive closures using the recursive capabilities of modern database systems. We believe that this is the first approach to use modern SQL capabilities for evaluating recursive predicates in logic-based query languages.

Chapter 1

Introduction

The project DDD[1] is a large effort of various research groups in the field of linguistics, history, and literature 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 texts will be annotated with respect to the structure of their content as well as with bibliographic and other meta-data. While much of this annotation is flat, such as a part-of-speech value attached to a word, a considerable amount of the annotation is highly structured. In particular, the syntactic structure of sentences is described by using complex data structures. Usually, these data structures are restricted to trees. An example of a syntax tree from the Penn Treebank[2] is given in Fig. 1.

However, in many cases linguists are interested in annotating more complex relationships between the words and grammatical substructures of a sentence. Such relationships cannot be modelled as trees any more, because they may contain cyclic dependencies, especially in the presence of ambiguous interpretations of a sentence. For instance, Fig. 2 depicts the structure of a German sentence together with information about the dependencies between certain grammatical substructures and the subject of the sentence. The resulting graph of all relationships may be cyclic since these dependencies may refer to arbitrary subtrees [1].

Non-tree syntax representations in linguistic databases have so far been mostly ignored, with the exception of the TigerSearch Engine [1]. In TigerSearch, such structures are stored as trees, while non-tree edges are managed by a special mechanism and require a special syntax in queries.

[1] `www.deutschdiachrondigital.de`

[2] See http://www.cis.upenn.edu/ treebank/home.html

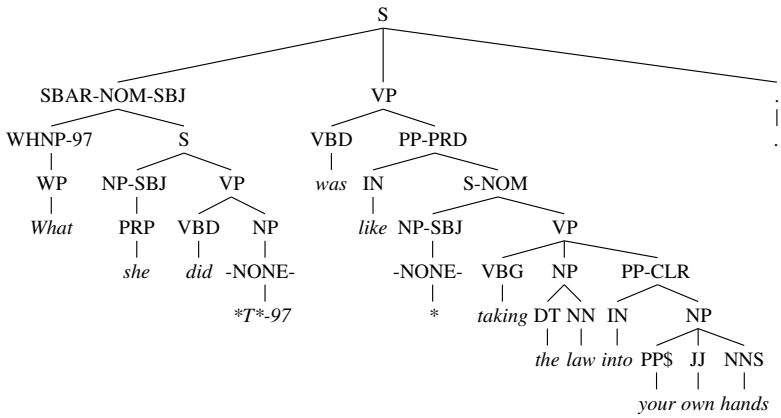

Figure 1.1: Syntactic structure of the sentence “What she did was like taking the law into her own hand” (from [1], p.6).

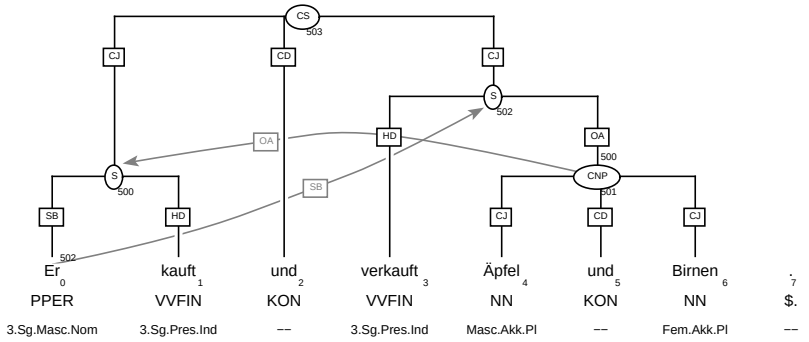

Figure 1.2: Structure of the sentence “Er kauft und verkauft Äpfel und Birnen.”, meaning “He buys and sells apples and pears.” (from [1], p.9).

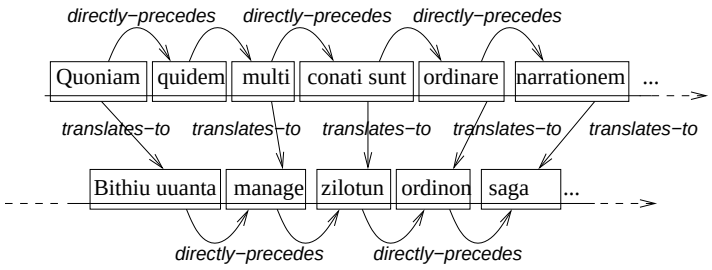

Figure 1.3: Alignment of corresponding phrases in a bilingual Latin / Old High German text (Tatian), represented as a graph over text spans.

In the DDD project, we are developing methods to store and manage large collections of richly annotated historic texts in a relational database [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 [3]. For instance, the logical organisation of a text in sections, paragraphs, sentences, and words often conflicts with the structure of its physical source (typically a book) in pages, lines, and whitespace-separated groups of characters (sentences may cross several lines, logical words may be hyphenated etc.). Moreover, the research interests of historical linguists require texts to be represented as several parallel text layers (e.g., a so-called *diplomatic* version close to the physical source, a more readable *normalised* version, a word-by-word translation, alternative versions from different text witnesses etc.) which need to be carefully aligned with each other. Alignments are represented as graphs over text spans (i.e., substrings of text layers), see Fig. 3.Together with spatial relationships between the spans of a text layer (precedence, inclusion, intersection etc.) and different annotation layers on top of these text layers, this complex representation poses further challenging requirements to the query language.

For these reasons, we believe that using XML and related query languages such as XPath or XQuery is not the best choice for modelling and querying the DDD corpus. Instead, we develop a logic-based query language whose queries are translated into SQL commands executed by a commercial relational database management system.

In this paper, we present a generic framework for developing a query language for a given application domain (here linguistics) with capabilities to query cyclic structures. In our approach, a query language consists of a set of logic predicates. For each predicate, one or more templates are defined in advance by the language designer. Each template of a predicate specifies SQL expressions for computing values for the output parameters of this predicate and SQL conditions on its input parameters. Our approach enables database designers or even domain experts with some SQL expertise to design domain-specific query languages.

Given a query as a logical combination of predicates, a query translation algorithm combines the different fragments of the predicates into a single SQL command that is executed by a relational DBMS, thus computing the answer to the query. There are two main differences to previous approaches. First, each query is translated into a single SQL command, thus leveraging the query optimization capabilities of commercial database management systems. Second, queries may contain a certain form of recursion, and are nevertheless translated into a single query. To translate such queries, we have developed the LoToS (Logic To SQL) query compiler that leverages the support for recursive queries now available in commercial database systems.

1.1 Running Example

In [2] we have presented predicates for a linguistic query language. Since the query compiler presented here is generic and the complexity of an application domain is encapsulated in the domain-specific set of predicates, we do not enter further the intricate domain of richly annotated historic corpora. (However, some examples from [2] are shown in the Appendix, together with the translations produced by LoToS.) Instead we use uninterpreted directed labelled graphs as a running example. The underlying relational schema consists of the tables `vertex(id,label)` and `edge(source,label, target)` where `edge.source` and `edge.target` reference `vertex.id` as foreign keys.

This schema is reflected by the predicates *vertex*() and *edge*(). Their extensions are defined by the corresponding tables:

$$\begin{aligned} \mathit{vertex}(V,L) &\equiv \langle V,L\rangle \in \texttt{vertex} \\ \mathit{edge}(S,L,T) &\equiv \langle S,L,T\rangle \in \texttt{edge} \end{aligned}$$

We use horn clauses to denote queries. For instance, the query

$$q(V,L) \leftarrow \mathit{vertex}(V,L)$$

retrieves all vertices and their labels. It can be expressed in SQL as:

```
SELECT id, label
FROM vertex
```

The challenge that we meet in this paper is how to translate nontrivial queries consisting of several predicate calls with shared variables under binding constraints imposed by the predicate definitions into closed SQL queries.

1.2 Structure of the Paper

In the next section, we describe the query translation algorithm without recursive predicates, which are introduced in Section 3. Section 4 discusses related work and Section 5 concludes the paper.

Chapter 2

The Compilation Algorithm

Before we present the compilation algorithm, we motivate and formulate some requirements to be met by the query compiler.

2.1 Requirements

2.1.1 Binding-dependent translation.

The translation of a parameter call in a query depends on which of its parameters are bound. For instance, the translation of query

$$q(V) \leftarrow vertex(V, \texttt{"foo"})$$

that searches for all vertices labeled `"foo"` needs to take into account that the second parameter of predicate *vertex*() is bound, by specifying only a single column in the `SELECT` clause and by adding an appropriate condition to the `WHERE` clause:

```
SELECT v.id
FROM vertex v
WHERE v.label= "foo"
```

Requirement: *The predicate definition language must provide a mechanism to specify alternative binding-dependent translations.*

Other predicates cannot be called at all with any unbound arguments. For instance, to support wildcard search in labels, the SQL string pattern matching operator `(. LIKE .)` is used to define a predicate $like(S,P) \equiv (S \texttt{ LIKE } P)$. This predicate can be used in the query $q(V) \leftarrow vertex(V,L) \wedge like(V, \texttt{"a\%"})$ to retrieve all vertices the label of which starts with "a". In SQL this query can be expressed by:

```
SELECT v.id
FROM vertex v
WHERE v.label LIKE "a%"
```

However, the query $q(S) \leftarrow like(S, \texttt{"a\%"})$ cannot be translated to SQL since it is unsafe: its extension is the set of all strings S such that (S `LIKE "a%"`) is satisfied.

Requirement: *The translation algorithm must detect illegal binding patterns.*

2.1.2 Shared variables.

Non-trivial queries involve several interdependent predicate calls that are combined by logical connectors. Dependencies between predicate calls are specified by using shared variables. These variables act as channels for information flow from one predicate (that binds a variable) to another (that uses this variable). Moreover, more than one predicate sharing a variable may provide a binding for this variable. In this case it must be ensured that all bindings are equal. In SQL, this amounts to introduce appropriate equality constraints.

Consider for instance the query

$$q(U,V) \leftarrow vertex(U, \texttt{"a"}), edge(U, \texttt{"b"}, V), vertex(V, \texttt{"c"})$$

that retrieves all pairs $\langle U,V \rangle$ of vertex identifiers such that the vertex identified by U is labeled `"a"`, the vertex identified by V is labeled `"c"`, and an edge labeled `"b"` connects the first with the second vertex. This query can be translated into SQL as follows:

```
SELECT u.id, v.id
FROM vertex u, edge e, vertex v
WHERE u.label= "a"
AND e.label="b"
AND v.label="c"
AND u.id= e.source
AND e.target= v.id
```

Variable U is computed both by the SQL expression `u.id` and by `e.source`. Therefore these alternative implementations must be connected by the equality constraint `u.id=e.source`. Similarly, variable V is computed both by `v.id` and `e.target` which need to be equated in the `WHERE`-clause. The information flow from the query body to the query head where U and V are used is implemented by including the expressions `u.id` and `v.id` in the `SELECT` clause. Which particular SQL expression from a set of alternatives is chosen for a variable does not matter because all alternatives are connected by equality constraints.

Requirement: *Information flow via shared variables must be implemented by propagating variable values in form of SQL expressions and by introducing appropriate equality constraints.*

2.2 Preliminaries

A query q is defined as a horn clause $H \leftarrow F$. The head $H = h(x_1, \ldots, x_m)$ defines the variables $x_1, \ldots, x_m$ as output parameters of q. F is a Boolean formula defined recursively as $F ::= F_1 \vee F_2 | F_1 \wedge F_2 | \neg F_1 | p(t_1, \ldots, t_n)$. h and p are predicate symbols. The call parameters t_j are either constants or variables. 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 into their definition before query translation, as described later. A primitive predicate p is defined by one or more templates T, each of which provides a SQL implementation for a certain binding pattern that can be instantiated to a SQL SELECT statement. Primitives are not necessarily extensional database predicates since their templates may combine data from multiple tables of the underlying database.

Definition 2.2.1 (Template)
A template T for a predicate $p(a_1, \ldots, a_m)$ has the form $(A, I, R, \sigma, \tau, w)$ 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 a SQL condition that must be satisfied for each solution of p.

□

Example 2.2.1

The template for predicate $vertex(U,L)$ may be defined as

$$T_{vertex} = (\langle U,L\rangle, \emptyset, \{\mathtt{v}\}, \{U = \mathtt{v.id}, L = \mathtt{v.label}\}, \{\mathtt{v}) = \mathtt{vertex}\}, \text{true})$$

A call to predicate *vertex*, for instance $vertex(\texttt{"v21"}, L)$, induces a binding β for all (here: zero) input parameters and some output parameters, i.e., $\beta(U) =$ `"v21"`. The template can be expanded then to the following SQL query which retrieves the label for vertex `"v21"`:

```
SELECT v.label
FROM vertex v
WHERE v.id= "v21"
```

□

Example 2.2.2

In contrast, the template for predicate like(S,P) requires S and P to be bound since they both occur in the range of the input substitution:

$$T_{vertex} = (\langle S,P\rangle, \{S,P\}, \emptyset, \emptyset, \emptyset, (S \texttt{ LIKE } P))$$

□

A single template $T = (A,I,R,\sigma,\tau,w)$ can be translated to a SQL SELECT statement given a substitution $\beta : B \to \mathscr{E}_{R,\emptyset}$ that assigns ground SQL expressions to a set B of parameters including all input parameters (i.e., $I \subseteq B \subseteq A$). The column expressions are formed by the SQL expressions assigned by σ to all free output parameters, i.e., $\{e_1,\ldots,e_k\} = \{\beta(\sigma(a)) | a \in O - B\}$. Those output parameters bound by β result in a set of equality constraints $\{c_1,\ldots,c_l\} = \{(\beta(a) = \beta(\sigma(a))) | a \in O \cap B\}$. For instance, in Example 2.2.1, there is a single condition `v.id= "v21"` since both σ and β are defined for parameter V.

The general form of the resulting SELECT statement is then:

```
SELECT e1, ..., ek
FROM τ(r1) r1, ..., τ(rn) rn
WHERE β(w) AND c1 AND ... AND cl
```

The challenge which we meet here is not to translate a single template, but to translate a whole query by combining templates for the predicates in the query into a closed SQL statement. This task requires some preprocessing which is discussed next.

2.3 Preprocessing

The body F of a query $H \leftarrow F$ is preprocessed by recursively expanding all macro calls. The result is transformed into a disjunctive form defined recursively by

$$\begin{aligned} F &::= F_1 \vee F_2 | C \\ C &::= C_1 \wedge C_2 | \neg C_1 | G \\ G &::= p(t_1, \ldots, t_n) \end{aligned}$$

Note that this is not disjunctive normal form since $\neg(C_1 \wedge C_2)$ is allowed. This is due to the fact that binding constraints must be respected. For instance, if predicate q in $p(X) \wedge q(X)$ requires its argument X to be bound and p provides these bindings, then $\neg(p(X) \wedge q(X))$ should not be transformed into $(\neg p(X)) \vee (\neg q(X))$ since q cannot be called with unbound X. (In contrast, $\neg(C_1 \vee C_2)$ is rewritten as $(\neg C_1) \wedge (\neg C_2)$ as expected.) We assume that every conjunction C is reordered into the form $G_1 \wedge G_r \wedge \neg C_1 \wedge \ldots \neg C_s$.

2.4 Translation

A query $q \equiv (H \leftarrow F)$ with head $H = h(x_1, \ldots, x_m)$ is translated into a SQL query Q that produces tuples with m attributes each. Every tuple $\langle c_1, \ldots, c_m \rangle$ in the result of Q yields a solution $\phi = \{x_1 = c_1, \ldots, x_m = c_m\}$ for q.

After the preprocessing step the query body F has the form $C_1 \vee \ldots \vee C_n$ where all C_i are free of disjunctions. If the C_i translate to SQL queries Q_i for $i = 1, \ldots, n$, then F translates to Q_1`UNION ALL` $Q_2 \ldots$`UNION ALL` Q_n.

Each conjunction C_i has the general form $G_1 \wedge \ldots \wedge G_r \wedge \neg N_1 \wedge \ldots \neg N_s$ where each G_j is a predicate call $p_j(t_{j1}, \ldots, t_{jn_j})$ and $N_1 \ldots N_s$ are again conjunctions of this general form (i.e., nested negations are allowed, c.f. Sec. 2.3). Let $V = \{x_1, \ldots, x_m, \ldots, x_n\}$ be the set of all free variables in $H, G_1, \ldots, G_r$ (some of which may be shared by $N_1 \ldots N_s$).

Since we do not allow function symbols, the predicate arguments t_{ji} are either variables from V or constants. We denote this domain of variables and constants by $\mathscr{T}_V$.

For each predicate call $G_j = p_j(t_{j1}, \ldots, t_{jn_j})$, $j = 1, \ldots, r$ let $T_j = (A_j, I_j, R_j, \sigma_j, \tau_j, w_j)$ be a template defining the called predicate p_j. Without loss of generality we assume that all parameters and table aliases in different templates T_j are disjoint. Note, that several templates may be defined for a predicate in order to offer variants with different binding requirements or to provide alternative implementations. Hence there may be alternative selections of templates for a query. Not every possible template combination can be translated successfully because the

binding requirements of all templates must be compatible. Multiple translations raise the issue of finding an optimal translation (with respect to some cost measure) which could be tackled using standard query optimisation methods. We have not yet explored these possibilities of our query translator.

Each predicate call $p_j(t_{j1},\ldots,t_{jm_j})$ defines a *parameter binding*, i.e., a substitution $\beta_j : A_j \to \mathscr{T}_V$ that assigns each parameter a_{ji} a predicate argument $\beta_j(a_{ji}) = t_{ji}$ for $i = 1,\ldots,m_j$. Since all parameters are disjoint, we can define a global binding $\beta = \bigcup_{j=1}^{r} \beta_j$ on the joint parameter set $A = \{a_{ji} | j = 1,\ldots,r,\ i = 1,\ldots,m_j\}$. The global set of input parameters is then $I = \bigcup_{j=1}^{r} I_j$. On $O = A - I$, the set of output parameters, the global output substitution σ is defined as $\sigma = \bigcup_{j=1}^{r} \sigma_j$.

Let $R = \bigcup_{j=1}^{r} R_j = \{r_1,\ldots,r_{|R|}\}$ be the global set of all table aliases and $\tau = \bigcup_{j=1}^{r} \tau_j$ the resulting global table assignment. Let $w = w_1$ `AND` $\ldots$ `AND` w_r be the conjunction of all template conditions w_j.

The set $E(x)$ of all expressions $e \in \mathscr{E}_{R,V}$ that compute $x \in V$ is defined by $E(x) = \{\beta(\sigma(a)) \mid a \in O, \beta(a) = x\}$. The translation algorithm fails for a chosen combination of templates if $E(x) = \emptyset$ for some $x \in V$. Otherwise there exists a substitution $\rho : V \to \mathscr{E}_{R,V}$ such that $\rho(x) \in E(x)$ for all $x \in V$. The expression $\rho(x)$ may still contain other variables from V. To resolve a variable x to a ground SQL expression, ρ must be applied exhaustively which is only possible if there are no cyclic dependencies like $\rho(x) = y$, $\rho(y) = x$ in which case the translation fails. Otherwise the substitution $\rho^* : V \to \mathscr{E}_{R,\emptyset}$ that applies ρ exhaustively until all variables are resolved is well-defined. We extend σ to I by defining $\sigma(x) = x \in \mathscr{E}_{R,I}$ for all $x \in I$. Now the substitution $\phi : A \to \mathscr{E}_{R,\emptyset}$ which maps parameters to ground SQL expressions can be defined by $\phi = \rho^* \circ \beta \circ \sigma$.

Due to shared variables $x \in V$ there may be several alternative expressions in $E(x)$ for computing x. These alternatives must be connected by equations which act as join conditions between different table aliases. In addition, for output parameters $a \in O$ that are bound to constants (denoted $\mathrm{const}(\beta(a))$) an appropriate equation must be added to the WHERE clause of the resulting SELECT statement. These requirements are taken care of by a set $Q = \{q_1,\ldots,q_{|Q|}\}$ of equality constraints defined by

$$\begin{aligned} Q &= \{(\phi(a) = \phi(a')) \mid a,a' \in O, a \neq a', \beta(a) = \beta(a')\} \\ &\cup \{(\phi(a) = \beta(a)) \mid a \in O, \mathrm{const}(\beta(a))\} \end{aligned}$$

Since the negative conditions $N_1,\ldots,N_s$ of C_i are again conjunctions of the same form as C_i, they can be translated into correlated sub-queries by a recursive call of the translation algorithm that takes substitution ρ^* into account. Let $S_1,\ldots,S_s$ be translations of $N_1,\ldots,N_s$.

With these definitions the conjunction C_i is now translated into the SQL statement

SELECT $\rho^*(x_1),\ldots,\rho^*(x_m)$
FROM $r_1,\ldots,r_{|R|}$
WHERE $\phi(w)$
AND q_1 **AND** ... **AND** $q_{|Q|}$
AND NOT EXISTS (S_1) **AND** ... **AND NOT EXISTS** (S_s)

2.5 Expressions

In many queries, arithmetic expressions need to be formulated. There are two approaches to this problem. The first approach is logically clean but verbose. It requires each n-ary function to be defined as an $(n+1)$-ary predicate. For instance, the condition $Y = 2X + 1$ would be encoded as $\text{times}(2,X,T) \wedge \text{plus}(\text{T},1,\text{Y})$.

The second approach is more convenient and has been implemented in our query compiler. It introduces an equality predicate the arguments of which can be arithmetic expressions (i.e., terms). For instance, the mentioned condition can be formulated naturally as $X = 2*Y+1$. This approach applies not only to arithmetic operations on numbers, but extends to operations on other data types, e.g., to string operations like concatenation, etc.

In general, an equality condition $t_1 = t_2$ where both t_1 and t_2 are terms containing free variables $v_1,\ldots,v_n$, is translated into a template

$$(\langle v_1,\ldots,v_n\rangle, \{v_1,\ldots,v_n\}, \emptyset, \emptyset, \emptyset, e_1 = e_2)$$

where e_1, e_2 are SQL expressions computing the expressions represented by term t_1, t_2, respectively. Definitions for the free variables $v_1,\ldots,v_n$ must be provided by other templates in the query.

If t_1 is a variable v_1, the resulting template is

$$(\langle v_1,\ldots,v_n\rangle, \{v_2,\ldots,v_n\}, \emptyset, \{v_1 = e_2\}, \emptyset, \text{true})$$

Thus the template assigns the SQL expression e_2 to v_1 which can be used as input variable in other templates then. The case where t_2 is a variable is analogous. If both t_1 and t_2 are variables, both possible templates are tried.

Tests other than equality (e.g., $<$, $\leq$) do not allow information flow between variables. They are supported by a predicate $\text{test}(t)$ which takes a term t representing a Boolean SQL expression b with free variables $v_1,\ldots,v_n$ as argument and translates it into the template

$$(\langle v_1,\ldots,v_n\rangle, \langle v_1,\ldots,v_n\rangle, \emptyset, \emptyset, \emptyset, b)$$

Example 2.5.1

The Boolean condition test$(X < Y)$ yields the template

$$(\langle X, Y \rangle, \{X, Y\}, \emptyset, \emptyset, \emptyset, (X < Y))$$

which can be used to translate a query containing this condition. □

2.6 Postprocessing: Self Join Optimisation

If two predicates that access the same table are combined in a conjunction, the resulting query may involve a redundant self-join of a table that can be avoided by unifying table aliases. For instance, let the convenience predicate edge_labels() be defined as a macro

$$\text{edge\_labels}(U, V, UL, EL, VL) \equiv \mathit{vertex}(U, UL) \wedge \mathit{edge}(U, EL, V) \wedge \mathit{vertex}(V, VL)$$

This macro retrieves edges together with the labels of its endpoints. The query

$$\begin{aligned} q(UL, VL, WL) \quad \leftarrow \quad & \text{edge\_labels}(U, V, UL, \_, VL) \ \wedge \\ & \text{edge\_labels}(U, W, UL, \_, WL) \ \wedge \\ V \neq W & \end{aligned}$$

(which returns the labels of two edges $U \rightarrow V$ and $U \rightarrow W$) translates to

```
SELECT v1.label, v2.label, v4.label
FROM vertex v1,
  edge e1,
  vertex v2,
  vertex v3,
  edge e2,
  vertex v4
WHERE v1.id=e1.source
AND e1.source=v3.id
AND v3.id=e2.source
AND v1.label=v3.label
AND e1.target=v2.id
AND e2.target=v4.id
AND e1.target <> e2.target
```

Since the table aliases v_1 and v_3 for table `vertex` are connected by the join conditions `v1.id=e1.source` and `e1.source=v3.id`, their keys `v1.id` and `v3.id` are equal. This means that v_1 and v_3 always reference the same tuple. By unifying the two aliases and removing redundant equations we can save one join operation:

```
SELECT v13.label, v2.label, v4.label
FROM edge e1,
  vertex v13,
  vertex v2,
  edge e2,
  vertex v4
WHERE e1.source=v2.id
AND v2.id=e2.source
AND e1.target=v1.id
AND e2.target=v3.id
AND e1.target <> e2.target
```

This optimisation requires two aliases for the same table for which it can be deduced from the equality constraints in the WHERE clause that their key attributes are equal. In this case, the aliases are unified and redundant equations are removed.

Chapter 3

Recursive Queries

Many application domains such as linguistics (see Sec. 1) and bioinformatics deal with data that is represented in form of graphs or restrictions thereof, such as directed acyclic graphs (DAGs) or trees. In bioinformatics for instance, graphs and graph queries play an increasingly important role to model and query metabolic networks [4].

An important class of queries in such application domains deals with paths in graphs. Paths can easily be specified using recursive predicates. For instance, the (non-reflexive) transitive closure in a graph can be defined by

$$\mathrm{tc}(A,D) \quad \equiv \quad edge(A,L,T) \wedge (T = D \vee \mathrm{tc}(T,D))$$

Linear recursive queries have been introduced to SQL in the SQL:1999 standard and are now supported by some commercial database systems, such as Oracle and DB2. While DB2 supports recursive queries conforming to SQL:1999, Oracle offers a proprietary SQL extension for so-called *hierarchical queries* that amount to computing the transitive closure of a binary relation between rows of a table.

In hierarchical data, such as XML document trees, precomputed pre-/post-order ranks can be used to answer path queries much more efficiently compared to recursive SQL queries [5]. As shown in [6], this technique can be extended to DAGs.

Our goal is to support a class of recursive queries that (i) can be implemented on top of both DB2 and Oracle and (ii) can make use of pre-/post-order rank indexes where possible. This class consists of horn clauses extended with the (non-reflexive) transitive closure operator $(.)^+$, defined by

$$p(X_0,X_n)^+ \equiv \exists n > 0, X_1, \ldots, X_{n-1} : p(X_0,X_1) \wedge \ldots \wedge p(X_{n-1},X_n)$$

The reflexive transitive closure operator $(.)^*$ is syntactic sugar defined by

$$p(X_0,X_n)^* \equiv X_0 = X_n \vee p(X_0,X_n)^+$$

We assume that predicate p can be translated to a sub-query P with columns x,y. The call $p(X,Y)^+$ can be expressed in SQL:1999 as:

```
⟨X,Y⟩ ∈ (
WITH RECURSIVE r(x,y) AS (
   (  SELECT x,y
      FROM P p
      [ WHERE x=X ]X
   )  UNION ALL (
      SELECT r.x, p.y
      FROM P p, r
      WHERE r.y = p.x
   ))
SELECT DISTINCT *
FROM r
WHERE [y=Y]Y)
```

The notation $[B]_V$ indicates a code fragment B that is included if variable V is bound.

In Oracle SQL, the call $p(X,Y)^+$ can be expressed using a hierarchical query:

```
⟨X,Y⟩ ∈ (
SELECT (CONNECT_BY_ROOT x), y
FROM p
[WHERE y=Y ]Y
[START WITH x=X ]X
CONNECT BY PRIOR y = x)
```

If the graph $G = (V,E)$ induced by p (i.e., $V = \{u|\exists v : p(u,v)\} \cup \{v|\exists u : p(u,v)\}$, $E = \{(u,v)|p(u,v)\}$) is finite and acyclic, then a pre-/post-order rank index r_p can be computed. The fact $r_p(V,R_{\text{pre}},R_{\text{post}})$ states that node V has been reached in a depth first traversal at step R_{pre} and has been left at step R_{post} after traversing all of its descendents. Note, that the traversal algorithm may visit the children of a node in arbitrary order if no ordering criterion is defined, such as document order in an XML document tree. If the graph G is not a tree, then a node may be visited several times. For each visit, it receives a different tuple in r_p. If the graph has several root nodes, it is treated as if all roots were children of a virtual root.

If a pre-/post-order index r_p exists for a predicate p, then the call $p(X,Y)^+$ can be rewritten using the following equivalence:

$$p(X,Y)^+ \equiv r_p(X,X_{\text{pre}},X_{\text{post}}) \wedge r_p(Y,Y_{\text{pre}},) \wedge X_{\text{pre}} < Y_{\text{pre}} \wedge Y_{\text{pre}} < X_{\text{post}}$$

The query compiler translates this call as follows:

```
SELECT x.node,y.node
FROM r_p x, r_p y
WHERE x.pre < y.pre
AND y.pre < y.post
[ AND x.node= X ]X
[ AND y.node= Y ]Y
```

Example 3.0.1

Given a macro definition $edge(S,T) \equiv edge(S,\_,T)$, the query

$$q(V_1,V_2) \leftarrow vertex(V_1,a) \wedge edge(V_1,V_2)^+ \wedge vertex(V_2,b)$$

retrieves all pairs of a vertex V_1 labeled `"a"` and a vertex V_2 labeled `"b"` that are connected by a path from V_1 to V_2. This is translated to the SQL:1999 statement

```
WITH RECURSIVE tc(source,target) AS (
  ( SELECT edge.source, edge.target
    FROM vertex, edge
    WHERE vertex.id = edge.source
    AND vertex.label="a"
  ) UNION ALL (
    SELECT tc.source, edge.target
    FROM tc, edge
    WHERE tc.target= edge.source
  ))
SELECT tc.source, tc.target
FROM tc, vertex
WHERE tc.target= vertex.id
AND vertex.label="b"
```

For Oracle, this query is translated to a SELECT statement with a hierarchical SELECT statement as inline view `tc` that is correlated with the vertex tuples `v1` and `v2` by the `START WITH` and `WHERE` conditions. Note that the `WHERE` clause in the sub-query is evaluated on the *last* path element, i.e., the edge pointing to V_1.

```
SELECT  v1.id,  v2.id
FROM vertex v1,
  ( SELECT 1
    FROM
      ( SELECT e.source AS source, e.target AS target
        FROM edge e ) s
    WHERE v2.id=s.target
    START WITH v1.id=s.source
    CONNECT BY NOCYCLE (PRIOR s.target)=s.source
  ) tc, vertex v2
```

```
WHERE v1.label="a"
AND v2.label="b";
```

□

Note, that predicate *p* may have a more complex definition than just a single table. In SQL:1999, two calls of *p* are translated in the `WITH RECURSIVE` part, one in the initializing `SELECT` statement, and the other in the recursive `SELECT` statement.

In the case of Oracle SQL, the call of *p* is translated separately as a sub-query that is included as inline view in the hierarchical query.

Chapter 4

Related Work

Principles of Datalog and its evaluation are presented in [7]. The NAIL! system [8], an early deductive database system also translated programs written in Nail into SQL. In [9] the system (and the approach in general) is criticized for producing inefficient SQL code that required a large number of temporary tables and for its lack of control over the optimisation of the generated SQL queries.

Other deductive database prototypes that used relational database systems as back-ends are LOLA [10], DECLARE and SDS [11], and LDL++ [12].

The translation of GraphLog queries into SQL queries for DB2 are discussed in [13]. Recursive graph log queries are translated into recursive views in DB2.

In [14] a query compiler is presented that translates a query over a set of datalog rules into a C-program with embedded SQL. The generated program fetches tuples from intermediate queries from the database and sends them back to be inserted in temporary tables which causes a large communication overhead. The advances of commercial DBMS since 1998 allow LoToS to delegate more work to the database system.

There have been many proposals for query languages on linguistic data. However, most languages were developed for small corpora and use main-memory based evaluation schemes. A detailed overview can be found in [2]. More recently, XML has been proposed as the fundamental structure to store annotated corpora [15, 16]. However, as stated previously, the DDD annotation contains structures not easily modelled or queried in XML. Therefore, we believe that our approach is a more powerful alternative, although its performance still needs to be evaluated.

Chapter 5

Conclusion and Future Work

Domain-specific query languages are useful in various application domains, including complex linguistic databases [1, 15, 16] or metabolic network databases in biology [4].

The LoToS query compiler presented in this work enables the rapid implementation of domain-specific query languages on top of a relational database system with a fixed schema. The resulting logic-based query language provides a clean semantics and independence from particular data modelling decisions or vendor-specific SQL extensions. These advantages also make it a good target language for compilation of queries in a more concise query language in case that the language of logic formulas is found too verbose in a particular application domain.

As already mentioned, we have not yet explored the possibilities of query optimisation in our approach. Cost based optimisation should be used to chose between different translation generated by different predicate templates. Furthermore, the technique described in 2.6 should be considered as a special case of query minimization, i.e., techniques to rewrite a complex query in a simpler, yet semantically equivalent query [17]. However, many query minimization techniques are only applicable under set semantics and thus not in our scenario.

A prototype of the LoToS query compiler implemented in SWI Prolog is available[1].

[1] `http://www.informatik.hu-berlin.de/~faulstic/projects/DDD/software/LoToS`

Bibliography

[1] Lezius, W.: Ein Suchwerkzeug für syntaktisch annotierte Textkorpora. PhD thesis, Institut für maschinelle Textverarbeitung (IMS), Universität Stuttgart (2002)

[2] Faulstich, L.C., Leser, U., Lüdeling, A.: Storing and querying historical texts in a relational database. Informatik-Bericht 176, Institut für Informatik, Humboldt-Universität zu Berlin (2005)

[3] 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)

[4] Leser, U.: A query language for biological networks. Informatik-Bericht 187, Institut für Informatik, Humboldt-Universität zu Berlin (2005)

[5] Grust, T., Keulen, M.V., Teubner, J.: Accelerating XPath evaluation in any RDBMS. ACM Transactions on Database Systems **29** (2004) 91–131

[6] Trissl, S., Leser, U.: Querying ontologies in relational database systems. In: 2nd Conference on Data Integration in the Life Sciences (DILS05). (2005)

[7] Ullman, J.D.: Database and Knowledge-Base Systems, Vol. 2. Computer Science Press, Maryland (1990)

[8] Morris, K., Ullman, J.D., Gelder, A.V.: Design overview of the NAIL! system. In: Proc. of Third intl. conf. on logic programming. (1986) 554–568

[9] Derr, M.A., Morishita, S., Phipps, G.: The glue-nail deductive database system: design, implementation, and evaluation. The VLDB Journal **3** (1994) 123–160

[10] Zukowski, U., Freitag, B.: The deductive database system LOLA. In Dix, J., Furbach, U., Nerode, A., eds.: Proceedings of the 4th International Conference on Logic Programing and Nonmonotonic Reasoning. Volume 1265 of LNAI., Berlin, Springer (1997) 375–386

[11] Kießling, W., Schmidt, H., Strauß, W., Dünzinger, G.: Declare and sds: early efforts to commercialize deductive database technology. The VLDB Journal **3** (1994) 211–243

[12] Arni, F., Ong, K., Tsur, S., Wang, H., Zaniolo, C.: The deductive database system LDL++. Technical Report cs/0202001, arXiv (2002)

[13] Eigler, F.C.: Translating graphlog to SQL. In: CASCON '94: Proc. of the 1994 conf. of the Centre for Advanced Studies on Collaborative research, IBM Press (1994) 14

[14] Sunderraman, R., Sunderraman, R.: A deductive rules processor for SQL databases. In: ACM-SE 36: Proceedings of the 36th annual Southeast regional conference. (1998) 64–73

[15] Bird, S., Buneman, P., Tan, W.C.: Towards a query language for annotation graphs. In: 2nd intl. Conf. on Language Resources and Evaluation (LREC 2000). (2000) 807–814

[16] 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)

[17] Kunen, I.K., Suciu, D.: A scalable algorithm for query minimization. Technical Report 02-11-04, University of Washington (2002)

Appendix

As illustration to the compilation method demonstrated in the paper, we present some linguistic queries from [2] together with the translations produced by the LoToS query compiler. For definitions of predicates and for the underlying relational schema please refer to this report.

5.1 Searching for word forms

Sentences s where verb "sagen" occurs in second person singular This query combines a condition on the logical text structure (a token t within a sentence s) with conditions on the lemma annotation (lemma name n equals "sagen") and the inflectional morphology f.

$$
\begin{aligned}
Q_a(S) \equiv\ & \text{element}(\texttt{'Sentence'}, S) \wedge \text{element}(\texttt{'Token'}, T) \wedge \\
& \text{ancestor}(S, T) \wedge \\
& \text{element}(\texttt{'Lemma'}, L) \wedge \text{parent}(L, T) \wedge \\
& \text{element}(\texttt{'Entry'}, E) \wedge \text{parent}(L, E) \wedge \\
& \text{element}(\texttt{'LemmaName'}, N) \wedge \text{parent}(E, N) \wedge \\
& \text{elementSpan}(N, S_n) \wedge \text{string}(S_n.\texttt{tid}, \texttt{'sagen'}, S_n) \wedge \\
& \text{element}(\texttt{'FlexMorph'}, F) \wedge \text{parent}(F, T) \wedge \\
& \text{element}(\texttt{'Verb'}, V) \wedge \text{parent}(F, V) \wedge \\
& \text{attribute}(V, \texttt{'person'}, 2) \wedge \text{attribute}(V, \texttt{'number'}, \texttt{'sing'})
\end{aligned}
$$

This query is translated to:

```
SELECT
  element1.id AS Sentence
FROM
  element element1,
  element element2,
  rank ancestor,
```

```
  rank descendent,
  element element3,
  rank child1,
  element element4,
  rank child2,
  rank child3,
  element element5,
  text,
  element element6,
  rank child4,
  element element7,
  rank child5,
  attribute attribute1,
  attribute attribute2
WHERE element1.name="sentence"
AND element2.name="token"
AND element3.name="lemma"
AND element4.name="entry"
AND element5.name="lemma_name"
AND SUBSTR(text.content,
           element5.span.left,
           element5.span.right-element5.span.left)="sagen"
AND element6.name="flex_morph"
AND element7.name="verb"
AND attribute1.name="person"
AND attribute1.value=2
AND attribute2.name="number"
AND attribute2.value="singular"
AND element1.id=ancestor.element
AND element2.id=descendent.element
AND descendent.element=child1.element
AND child1.element=child4.element
AND element3.id=child1.parent
AND child1.parent=child2.parent
AND element4.id=child2.element
AND child2.element=child3.parent
AND child3.element=element5.id
AND element6.id=child4.parent
AND child4.parent=child5.parent
AND element7.id=child5.element
AND child5.element=attribute1.element
AND attribute1.element=attribute2.element
AND ancestor.pre=<descendent.pre
```

```
AND descendent.pre<ancestor.post
AND (element5.span IS NOT NULL)
AND element5.span.tid=text.id;
```

5.2 Querying aligned texts

How is "pulcher" (lat.) translated into Old High German? Query Q_b binds variable s_g to all spans that are aligned in role `'goh'` (i.e., German Old High) with a span in role `'lat'` that contains "pulcher" as content of a token t.

$$
\begin{aligned}
Q_b(S_g) \equiv\ & \text{element}('\texttt{Token}', T) \wedge \text{elementSpan}(T, S_t) \wedge \\
& \text{string}(S_n.\texttt{tid}, '\texttt{pulcher}', S_n) \wedge \\
& \text{element}('\texttt{Align}', A_l) \wedge \text{attribute}(A_l, '\texttt{role}', '\texttt{lat}') \wedge \\
& \text{elementSpan}(A_l, S_l) \wedge \text{contains}(S_l, S_t) \wedge \\
& \text{parent}(L, A_l) \wedge \text{element}('\texttt{Link}', l) \wedge \\
& \text{parent}(L, A_g) \wedge \text{element}('\texttt{Align}', a_g) \wedge \\
& \text{attribute}(A_g, '\texttt{role}', '\texttt{goh}') \wedge \\
& \text{elementSpan}(A_g, S_g)
\end{aligned}
$$

The query is translated to:

```
SELECT
  element4.span AS SpanGOH,
  SUBSTR(text2.content,
         element4.span.left,
         element4.span.right-element4.span.left) AS ContentGOH
FROM
  element element1,
  text text1,
  attribute attribute1,
  element element2,
  rank child1,
  element element3,
  rank child2,
  attribute attribute2,
  element element4,
  text text2
WHERE element1.name="token"
AND SUBSTR(text1.content,
           element1.span.left,
```

```
           element1.span.right-element1.span.left)="pulcher"
AND element2.name="align"
AND attribute1.name="role"
AND attribute1.value="lat"
AND element3.name="link"
AND element4.name="align"
AND attribute2.name="role"
AND attribute2.value="goh"
AND attribute1.element=element2.id
AND element2.id=child1.element
AND child1.parent=element3.id
AND element3.id=child2.parent
AND child2.element=element4.id
AND attribute2.element=element4.id
AND (element1.span IS NOT NULL)
AND element1.span.tid=text1.id
AND (element2.span IS NOT NULL)
AND element2.span.tid=element1.span.tid
AND element2.span.left=<element1.span.left
AND element1.span.right=<element2.span.right
AND (element4.span IS NOT NULL)
AND element4.span.tid=text2.id;
```

5.3 Querying Linguistic Trees

The following sample queries are taken from [16]. To facilitate comparisons, the query identifiers and the XML representation used there are adopted here: words are represented by elements named with the part-of-speech information (e.g., noun = `N`, verb= `V`); phrases are represented by elements whose name ends in a `P` (noun phrase = `NP`, verb phrase = `VP` etc.).

Noun phrases *np* that immediately follow a verb *v*.

$$\begin{aligned} Q_1(NP,V) \;\equiv\; & \text{element}(\texttt{'V'},V) \wedge \text{element}(\texttt{'NP'},NP) \wedge \\ & \text{elementSpan}(V,S_v) \wedge \text{elementSpan}(NP,S_{np}) \wedge \\ & \text{immediatelyPrecedes}(S_v,S_{np}) \end{aligned}$$

```
SELECT
  element2.id AS NP,
  element1.id AS V
```

```
FROM
  element element1,
  element element2
WHERE element1.name="V"
AND element2.name="NP"
AND (element1.span IS NOT NULL)
AND (element2.span IS NOT NULL)
AND element1.span.tid=element2.span.tid
AND element1.span.right=element2.span.left;
```

Noun phrases *np* which are the rightmost descendent of a verb phrase *vp*:

$$\begin{aligned} Q_6(NP,VP) \equiv\ & \text{element}(\texttt{'VP'},VP) \wedge \text{element}(\texttt{'NP'},NP) \wedge \\ & \text{ancestor}(vp,NP) \wedge \\ & \text{elementSpan}(VP,S_{vp}) \wedge \text{elementSpan}(NP,S_{np}) \wedge \\ & \text{suffix}(s_{NP},S_{vp}) \end{aligned}$$

```
SELECT
  element2.id AS NP,
  element1.id AS VP
FROM
  rank ancestor,
  rank descendent,
  element element1,
  element element2
WHERE element1.name="VP"
AND element2.name="NP"
AND ancestor.element=element1.id
AND descendent.element=element2.id
AND ancestor.pre=<descendent.pre
AND descendent.pre<ancestor.post
AND (element1.span IS NOT NULL)
AND (element2.span IS NOT NULL)
AND element2.span.tid=element1.span.tid
AND element2.span.left>=element1.span.left
AND element2.span.right=element1.span.right;
```

Verb phrases *vp* comprised of a verb *v*, a noun phrase *np*, and a prepositional phrase *pp*:

$$
\begin{aligned}
Q_7(VP,V,NP,PP) \equiv\ & \text{element}(\texttt{'VP'},VP) \wedge \text{element}(\texttt{'V'},V) \wedge \\
& \text{element}(\texttt{'NP'},NP) \wedge \text{element}(\texttt{'PP'},PP) \wedge \\
& \text{ancestor}(VP,V) \wedge \text{ancestor}(VP,NP) \wedge \text{ancestor}(VP,PP) \wedge \\
& \text{elementSpan}(VP,S_{vp}) \wedge \text{elementSpan}(V,S_v) \wedge \\
& \text{elementSpan}(NP,S_{np}) \wedge \text{elementSpan}(PP,S_{pp}) \wedge \\
& \text{prefix}(S_v,S_{vp}) \wedge \text{immediatelyPrecedes}(S_v,S_{np}) \wedge \\
& \text{immediatelyPrecedes}(S_{np},S_{pp}) \wedge \text{suffix}(S_{pp},S_{vp})
\end{aligned}
$$

```
SELECT
  element1.id AS VP,
  element2.id AS V,
  element3.id AS NP,
  element4.id AS PP
FROM
  rank ancestor1,
  rank descendent1,
  rank ancestor2,
  rank descendent2,
  rank ancestor3,
  rank descendent3,
  element element1,
  element element2,
  element element3,
  element element4
WHERE element1.name="VP"
AND element2.name="V"
AND element3.name="NP"
AND element4.name="PP"
AND ancestor1.element=ancestor2.element
AND ancestor2.element=ancestor3.element
AND ancestor3.element=element1.id
AND descendent1.element=element2.id
AND descendent2.element=element3.id
AND descendent3.element=element4.id
AND ancestor1.pre=<descendent1.pre
AND descendent1.pre<ancestor1.post
AND ancestor2.pre=<descendent2.pre
AND descendent2.pre<ancestor2.post
AND ancestor3.pre=<descendent3.pre
AND descendent3.pre<ancestor3.post
```

```
AND (element1.span IS NOT NULL)
AND (element2.span IS NOT NULL)
AND (element3.span IS NOT NULL)
AND (element4.span IS NOT NULL)
AND element2.span.left=element1.span.left
AND element2.span.right=<element1.span.right
AND element2.span.tid=element3.span.tid
AND element2.span.right=element3.span.left
AND element3.span.tid=element4.span.tid
AND element3.span.right=element4.span.left
AND element4.span.tid=element1.span.tid
AND element4.span.left>=element1.span.left
AND element4.span.right=element1.span.right;
```