Storing and Querying Historical Texts in a Relational Database

Lukas C. Faulstich[1], Ulf Leser[1], und Anke Lüdeling[2]

[1]{faulstic,leser}@informatik.hu-berlin.de
Institut für Informatik, Humboldt-Universität zu Berlin
[2]Anke.Luedeling@rz.hu-berlin.de
Institut für deutsche Sprache und Linguistik,
Humboldt-Universität zu Berlin

28th February 2005

Abstract

This paper describes an approach for storing and querying a large corpus of linguistically annotated historical texts in a relational database management system.

Texts in such a corpus have a complex structure consisting of multiple text layers that are richly annotated and aligned to each other. Modeling and managing such corpora poses various challenges not present in simpler text collections. In particular, it is a difficult task to design and efficiently implement a query language for such complex annotation structures that fulfills the requirements of linguists and philologists. In this report, we describe steps towards a solution of this task. We describe a model for storing arbitrarily complex linguistic annotation schemes for text. The text itself may be present in various transliterations, transcriptions, or editions. We identify the main requirements for a query language on linguistic annotations in this scenario. From these requirements, we derive fundamental query operators and sketch their implementation in our model. Furthermore, we discuss initial ideas for improving the efficiency of an implementation based on relational databases and XML techniques.

Contents

Chapter 1

Introduction

This paper describes an approach for storing and querying a large corpus of linguistically annotated historical texts in a database. It is being developed in an interdisciplinary project of linguists of historical German, corpus linguists, computational linguists, and computer scientists. In the field of computer science, our work touches the areas storage/retrieval of XML in databases, Information Retrieval and Multimedia Databases.

1.1 DeutschDiachronDigital

DeutschDiachronDigital[1] (henceforth DDD) is an endeavor of about 15 German research groups to establish a large corpus of older German texts (described in more detail in [Lüdeling et al., 2005]). There exist various notions of "corpus"; DDD has adopted the following definition due to Sinclair[Sinclair, 1996]:

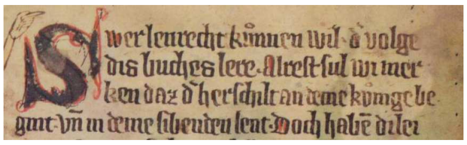
> A corpus is a collection of pieces of language that are selected according to explicit linguistic criteria in order to be used as a sample of the language [...] A computer corpus is a corpus which is encoded in a standardised and homogeneous way for open-ended retrieval tasks. Its constituent pieces of language are documented as to their origin and provenance.

The emphasis of DDD lies on creating a *diachronic* corpus that – in contrast to a *synchronous* corpus – documents the *evolution* of a language (or a group of languages) over a range of time, in our case from the earliest Old High German or Old Saxon texts from the 9th century up to Modern German at the end of the 19th century.

[1] `http://www.deutschdiachrondigital.de/`

Note that a linguistic corpus is more than a collection of transcribed texts: its main value arises from a rich system of annotations enabling automated statistical analyses and making texts accessible for research not only to specialists of particular dialects or language stages but also to a broader range of scholars such as historical linguists, theoretical linguists, philologists, historians, philosophers, etc. However, although many older texts (manuscripts and early prints) have been digitized and transcribed in a number of projects (for example, TITUS[2], Bibliotheca Augustana[3], MHDBDB[4]), a large linguistic corpus of older German is still missing (cf., [Kroymann et al., 2004]).

Historical texts are available in form of manuscripts such as the "Heidelberger Sachsenspiegel"[5] (cf. Fig. 1.1) or as early prints.



Swer lenrecht kůnnen wil. d' volge
dis bůches lere. Alreſt ſul wi mer
ken daz d' herſchilt an deme kůnge be
gunt. vn̄ in deme ſibenden lent. Doch habē [illegible]

Figure 1.1: Detail from page 1r of the "Heidelberger Sachsenspiegel".

A linguistic corpus of historical texts such as DDD must go well beyond digitization. It must organize texts in several aligned text layers together with a rich system of annotations. The workflow for producing this complex representation is discussed next.

1.2 Workflow

To add a text to the corpus, the following processing steps sketched in Fig. 1.2 have to be carried out. Most of them require manual work by experts with some computer support (mainly editor programs and annotation tools). All data (texts and annotations) are uploaded to the DDD-Server and stored in a database from where they can be retrieved for the next processing step.

1.2.1 Header

The first step is to produce a bibliographic description of the source manuscript (or original print) using available catalog information. This description together

[2]`http://titus.uni-frankfurt.de/`

[3]`http://www.fh-augsburg.de/~harsch/augusta.html`

[4]`http://mhdbdb.sbg.ac.at:8000/index.html`

[5]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`

with other metadata describing the text as a whole constitutes the so-called *header* of the text. The header has a complex structure and may contain very detailed information on the physical state, provenance, processing, encoding etc. of the physical source and its text (cf. [Sperberg-McQueen and Burnard, 2002] for the TEI Header format and the TEI Web site for the activities on Manuscript Description[6]).

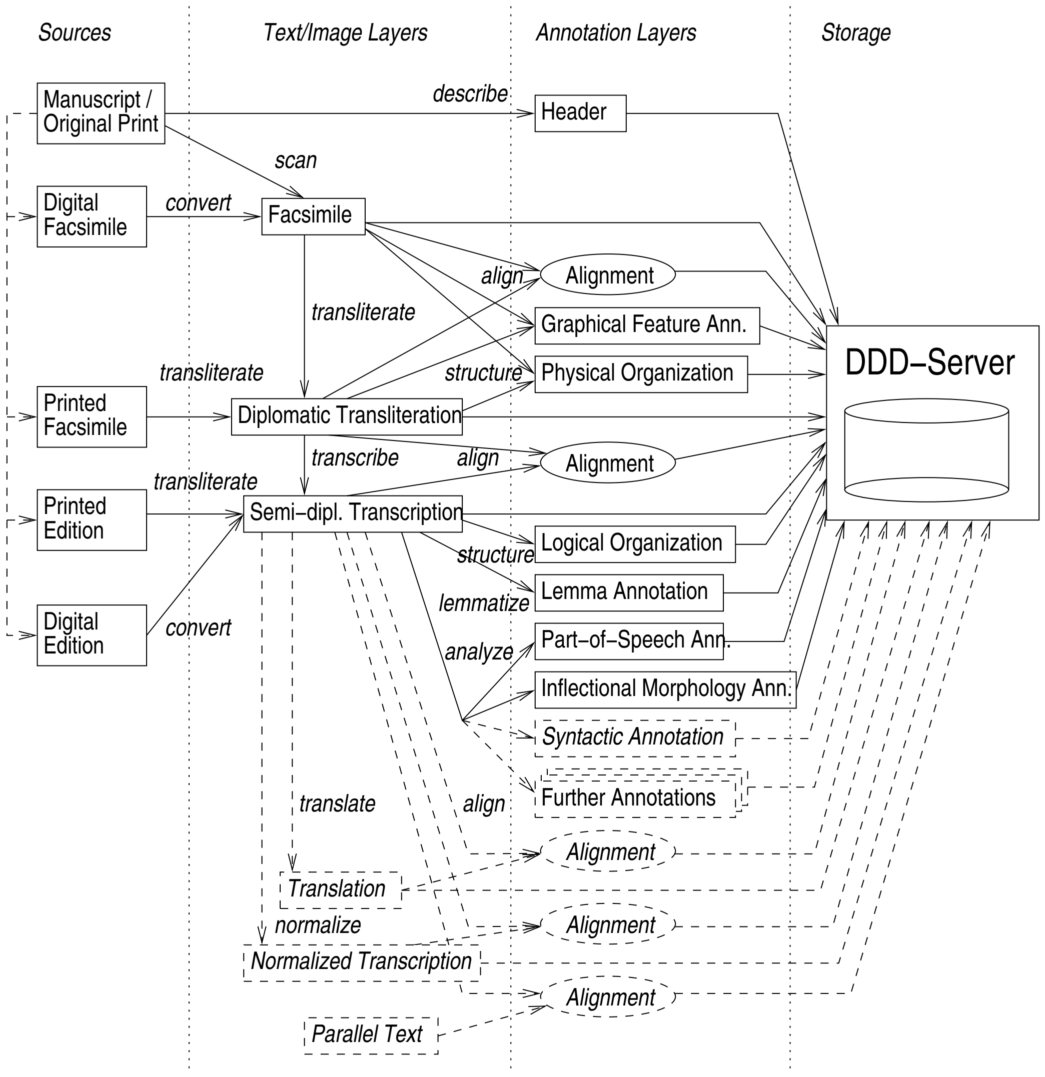


Figure 1.2: Production workflow of the DDD project.

[6]http://www.tei-c.org/Activities/MS/

1.2.2 Body

Diplomatic level. To produce the text *body*, a so-called *diplomatic* transliteration of the text is needed as a base text layer. In a linguistic context, *diplomaticity* refers to the closeness of an edition to the original manuscript. For instance, there often exist different variants of the same letter (so-called *allographs*). For instance, the small letter “s” has a round form used at the end of words and a long form used within words (c.f., the words “dis” and “alrest” in the second line of Fig. 1.1). These two forms are represented in the diplomatic layer using the Unicode characters `0073` and `017F`, respectively.

Several logical words may be written as one graphical word (e.g., in Fig. 1.1, “Swerlenrecht” = “Swer”+ “lenrecht”) while a single logical word may be distributed to several graphical words (e.g., in the case of a line break). Abbreviations are very common in manuscripts. The diplomatic layer tries to follow the original as closely as possible in representing graphical words and abbreviations.

There are several ways of how to arrive at the diplomatic text layer. Access to the manuscript itself or to a high-quality facsimile of it is indispensable. If digital facsimiles exist or can be produced, they will be made available to the user if possible, including an alignment of image regions with the corresponding text spans in the diplomatic text layer.

The diplomatic transliteration is an unstructured text encoded in the Unicode character set (extended with special characters in the *Private Use Area*). For instance, the “u” with a ring on top of it in the first line of Fig. 1.1 can be encoded in Unicode using the character “u” plus a combining diacritic “◦” (Unicode `030A`). Since manuscripts have many graphical features (such as ligatures, colors, initials, indentation, super-/sub-scripts etc.) that are important for a proper interpretation, an additional annotation layer documents graphical features visible in the manuscript and aligns them with text spans in the diplomatic text layer.

The physical organization of the text in pages, lines, and graphemic words is documented in a similar way as an annotation layer on top of the diplomatic transliteration.

Semi-diplomatic level. The diplomatic layer is then *transcribed* into a text layer we have termed “*semi-diplomatic*” by expanding abbreviations (e.g., *d˜* in the first line into *der*), splitting or connecting graphical words into logical words (tokens) etc. The logical structure of the text in chapters, sections, paragraphs etc. is added in form of annotations on top of this layer.

Linguistic annotations. Linguistic analyses such as lemma annotation, part-of-speech (e.g., noun, verb, adjective), inflectional morphology (e.g., third person singular, genitive plural), syntax trees, and possibly others, are further annotation layers based on this semi-diplomatic layer.

Other text layers. There may be several other layers, e.g., a normalized transcription (which uses a standardized spelling), a translation into Modern German, and alignments with parallel texts. For instance, the same text is sometimes available in several manuscripts which may be incomplete (due to damages and loss) and may differ in the overlapping parts. In this case, corresponding parts would need to be aligned with each other to enable comparative studies.

Conversion. Several partners will contribute to DDD electronic texts produced in earlier projects. For these texts conversion procedures need to be developed. The output of automated conversion scripts always needs to be proof-read against the manuscript and may need to be brought to a common standard corresponding to the rest of the DDD corpus. This may for instance require that a diplomatic version is created from a semi-diplomatic version by using facsimiles or the original manuscript.

Chapter 2

Requirements for Managing Linguistic Corpora

2.1 Data Model Requirements

Traditionally, linguistic corpora have been stored in unstructured files with rather simple formats (e.g., part-of-speech tagged corpora). SGML and XML encodings have been developed by the Text Encoding Initiative (TEI[1]. Databases are only rarely used to store linguistic corpora, e.g., in the TIGER project[2]. To support efficient search on a large text collection of approx. 100 million words, to ensure independence from external file formats, to ensure data integrity, and to manage a multitude of annotation levels together with multiple representations of text fragments, a relational database system for storing text and annotations is foreseen.

In contrast to typical information retrieval corpora, linguistic corpora store not only texts themselves, but also linguistic annotations, different aspects of a text (text layers) and alignments between and within text layers. Hence the main challenge in data modeling is to represent these annotations and their links with the underlying texts.

Many linguistic corpora use graphical words (tokens) as the smallest addressable unit. In historical texts, however, this is not sufficient since single characters need to be addressed. For instance, medieval manuscripts make heavy use of abbreviations, e.g. the character sequence "*er*" is often replaced by a so-called *title* (sic) symbol ("˜"), as in "*d̃* " (= "*der*" = Engl. 'the'). Such abbreviations will be expanded out in the normalized, unabbreviated word form. The alignment between the diplomatic and the semi-diplomatic text layer should associate an abbreviation with its expansion. To support this, alignments are modeled as a set of

[1][Sperberg-McQueen and Burnard, 2001], `http://www.tei-c.org`

[2][Brants et al., 2002], `http://www.ims.uni-stuttgart.de/projekte/TIGER/`

links between substrings of text layers (called *spans*). An alignment can associate spans from a single text layer or from different text layers.

An annotation is additional information that is attached to a span of a text layer. The simple-most type of annotation marks a span of text to make a statement about it, e.g., "this is a sentence". Annotations may have attributes to specify certain properties, e.g., "the grammatical case of this word is genitive, its grammatical number is plural". More complex annotations correlate different spans of text (e.g., for alignments, co-references) or build nested structures such as syntax trees. Annotations are organized into groups called *annotation layers*. An annotation layer describes a certain aspect of a text layer (e.g., its syntax) by a hierarchy of annotation elements corresponding to spans of this layer. Examples of annotation layers in our approach are presented in Sec. 1.2.2.

Corpora aim at representing all constituent texts in a uniform and comparable way to allow the computation of meaningful statistics. This means that a standard set of text and annotation layers is defined for every text. On the other hand, it is desirable to allow further layers to be added and to support extra layers on sub-corpora in order to accommodate specialized or unforeseen research needs.

It must be possible to express each annotation layer independently of the other layers. In general, each annotation layer can be represented as hierarchical (XML) markup of a text layer. Due to their independence, these markup hierarchies may be in conflict which prevents them from being merged into a single (XML) document hierarchy. For instance, lines and sentences may arbitrarily overlap in a text. In XML, this cannot be represented without resorting to tricks such as using empty "milestone" elements or splitting annotation elements at the boundaries of conflicting elements.

To cope with multiple text layers and conflicting hierarchies, *stand-off annotation* techniques have been developed (mainly in the field of multi-modal corpora) and have been standardized in XCES[Ide et al., 2000]. Stand-off annotation means to separate annotations from the underlying texts and use references (e.g. XLink/XPointer[3]) to specify the text spans or document elements to be annotated. This means that one has to distinguish internal edges (parent-child relationships within one annotation level) from external edges (links between different annotation levels or between annotations and text spans) when navigating a corpus annotated in this way. We require a separation of texts and annotations, too, but in addition want to avoid the distinction between internal and external edges. Since there can be cross-references between different texts within a corpus, the whole corpus needs to form a single complex data object.

Historical texts often have to be reconstructed from several partly contradicting manuscripts. Hence the data model must support the representation of such

[3]`http://www.w3.org/XML/Linking`

text variants.

Since texts and in particular historical texts often allow alternative linguistic interpretations, the data model must support the encoding of alternative annotations of the same text item. On the other hand, it must also allow for missing annotations since there are cases in which it is not possible to assign an annotation at all.

2.2 Requirements for Querying Annotated Texts

Fig. 2.1 shows a typical user session. The user enters a query, receives a list of results ("hit list") in form of a keyword-in-context (KWIC) concordance from which each result can be inspected by exploring the text surrounding the hit. In case the user needs to further analyze or edit the text using local tools or to present the result as an example in a publication, the result or the whole text can be downloaded in various formats.

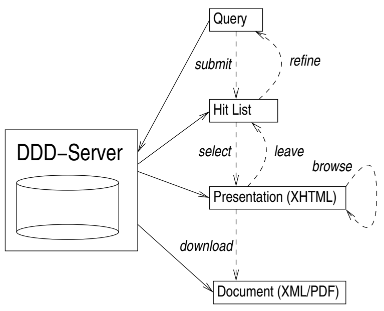


Figure 2.1: Typical user session.

Querying annotated texts hence comprises (i) formulating a query in an appropriate query language (ii) executing this query to search for relevant document elements and text spans and (iii) transforming the results into one of several presentation formats that can be displayed on a Web browser or downloaded for further analysis or processing.

2.2.1 Query Language

The query language should be intuitive for the users of the corpus, i.e., mainly linguists and philologists. Hence it would be convenient to build on existing corpus query languages such as TIGERSearch[Brants et al., 2002] or CQP[Christ, 1994]. The requirements for the expressiveness of the query language follow from the search requirements discussed next.

2.2.2 Complex Search Results

In contrast to conventional information retrieval, it is not sufficient to search just for whole texts. Rather users are looking for certain regions (*spans*) of a text. Within these spans, they are interested in certain elements matching our query. For instance, it is not sufficient to find all text spans matching a certain grammatical structure (e.g., a sentence with a relative clause), but on top of these spans the user might want to retrieve a pair of syntax tree nodes, one of which represents a sentence and the other a relative clause within this sentence. In addition, users might want aligned spans in other text layers to be retrieved as well.

2.2.3 Corpus Selection

Before a search within texts can be started, the user first has to define the corpus on which the search has to be performed. In most corpus search tools, the only choice of the user is between several pre-assembled corpora. Our approach is more flexible since it allows texts to be selected from the whole DDD corpus depending on conditions specified by the user. Such conditions can be posed for:

- Bibliographical data: texts in linguistic corpora typically have a very detailed header containing not only basic information such as title or authors, but providing much more data on the text as a whole such as details on the authors, writers, editors, on the preparation of the texts, used languages and dialects, genre, the social, historic, and geographic context etc.

- Automatically generated metadata: size of a text, existence of certain text or annotation layers, existence of aligned digital facsimile images etc. This is particularly important since the texts in the DDD corpus will be heterogeneous with respect to annotation depth and diplomaticity. A more homogeneous sub-corpus can be selected by specifying the minimum standard required for investigating a particular research question.

- Aggregated linguistic properties: e.g., the number of occurrences of certain lemmata or word forms or the frequency of certain linguistic features.

Sub-corpora can be assembled in advance by tagging each member text with a corpus identifier in its bibliographical data. For instance, one could prepare a sub-corpus of Middle High German Bavarian texts from tailored for the needs of a certain research area.

2.2.4 Single Text Spans

Text spans on a single text layer need to be searched by specifying

- literal substrings
- regular expressions
- intervals (e.g., characters $1 - 10,000$ of a text)

2.2.5 Single Elements

Single annotation elements should be searchable by their type (tag name), by conditions on their attribute values, and by position with respect to other elements such as "the third chapter element of text #4711" or "the first 100 word elements on page 17". Positions can be specified relative to the beginning or the end of some region that has to be specified within a subquery (e.g., "text layer #4711" or "page 17"); see the next section. Within this region, it needs to be specified what has to be counted for determining the position. For instance, one could count elements satisfying a certain attribute condition or elements of the same type as the result element.

2.2.6 Correlation of Spans and Elements

In queries it must be possible to combine conditions on the same items like "*a word element whose content matches the regular expression `/.*keit/`*" and to find items within other items, e.g., "*a word element within the page element representing page 31*". This is in particular important for querying syntax trees.

Sequences. Searches for sequences of items such as "*an article directly followed by a noun*" or "*the span between the first ',' and the final '.' within a sentence*" must be supported.

Use of alignments. A text may consist of several aligned text layers. This leads to the requirement of correlation between elements or spans over different text layers. Alignments need to be used to project spans to a common text layer where they can be compared (e.g., for containment). In addition, the user may want to align sequences of elements as well. Some example queries are:

1. *find all occurrences of the word 'der' (within the semi-diplomatic text layer) that correspond to a 'd˜' abbreviation (within the diplomatic text layer)*

2. *show all pairs of a line (an annotation on the diplomatic text layer) and an intersecting verse (an annotation on the semi-diplomatic text layer)*

3. *show all pairs of a Latin word form and a corresponding German word form in a bilingual (e.g, Latin, Old High German) text*

2.3 Transformation Requirements

A powerful and flexible method for transforming a result set into XML/HTML documents is needed to present search results to the user and to produce documents well-suited for printing or further processing. Requirements for this transformation method are discussed in [Dipper et al., 2004]. They include projection, selection, folding and rearrangements, deriving attributes, elements, and content, interchange between element content and attributes, context-sensitivity, identifier/URL generation, and encoding and decoding of conflicting hierarchies. This issue will be studied in future work.

Chapter 3

The ODAG Approach for Managing Linguistic Corpora

3.1 Architecture

DDD uses the Web-based three tier architecture on top of an object-relational database management system (ORDBMS) depicted in Fig. 3.1. The Web interface is used for searching, browsing, and downloading texts as well as for uploading new or revised texts to the database. Texts are edited and annotated using external XML-based tools. The Web interface offers import and export modules that convert between the internal representation in the database system and the particular XML formats of these tools.

When the user issues a query or navigates the HTML representation of a text, the user's Web browser sends HTTP requests to the Web interface. The application logic layer translates these requests into database queries. The results of these queries (i.e. tuples) are then transformed by one of the export modules into HTML which is returned to and then displayed by the Web browser. For download, documents in other formats such as PDF or TEI can be generated as well.

3.2 Data Model

In [Dipper et al., 2004], we have presented our logical ODAG (Ordered Directed Acyclic Graphs) datamodel which is shown in Fig. 3.2 as a class diagram. A Corpus consists of an open set of Text layers and a set of Nodes. A Node is either a Span in one of the Text layers or an Element. Each Element is described by a set of named Attributes. Optionally it refersTo a Span. The content of an Element consists of an ordered list of child Nodes. We abuse the UML conventions slightly by using an aggregation arrow for the $m : n$ relationship isChildOf to stress that

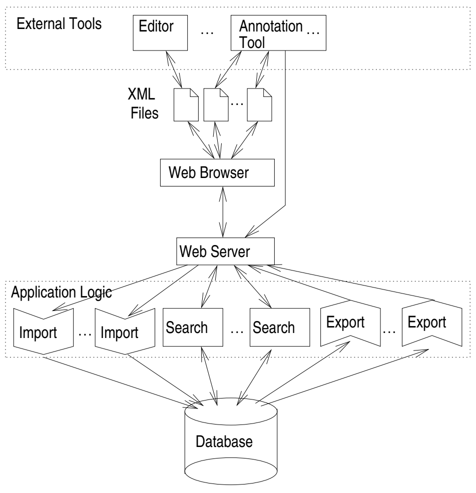


Figure 3.1: System Architecture of DDD

this relationship *aggregates* Nodes into an acyclic graph wherein each Node may have multiple parent Elements. This ODAG must have an unique root Element reachable from the Corpus object via the hasRoot association.

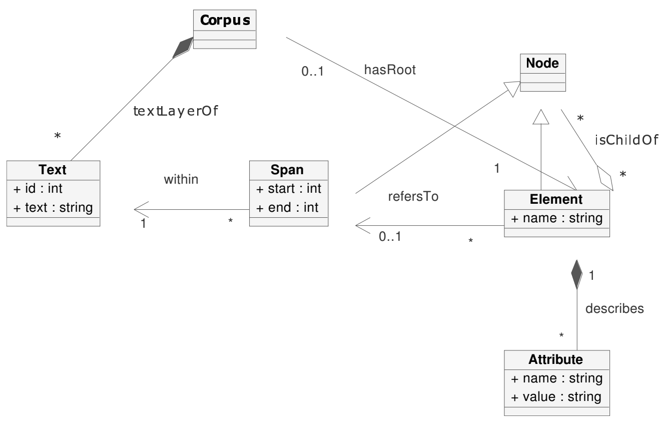


Figure 3.2: Meta-model of the DDD datamodel in form of an UML class diagram.

Just like DTDs or XML schemata specify the structure of conforming XML documents, an *ODAG schema* specifies a class of ODAGs (called instances of the schema) whose structure conforms to the schema.

For a particular corpus such as DDD, an ODAG schema needs to be developed to specify the structure of the corpus. The schema will be the result of a comprehensive standardization effort involving all participating groups.

Currently we are developing a preliminary schema for DDD on prototypical texts. Fig. 3.3 sketches a detail of such a text, the *Sachsenspiegel* (cf. Fig. 1.1).

The diagram shows the Physical hierarchy of the text (Volume / Page / Line / (graphemic) Word) as annotations of the diplomatic text layer as well as the Logical hierarchy of the text (Part / Chapter / Section / Paragraph / Sentence / Token (=logical word)) as annotation of the semi-diplomatic text layer. Each element in these hierarchies refers to a span on the underlying text layer (indicated by an arc).

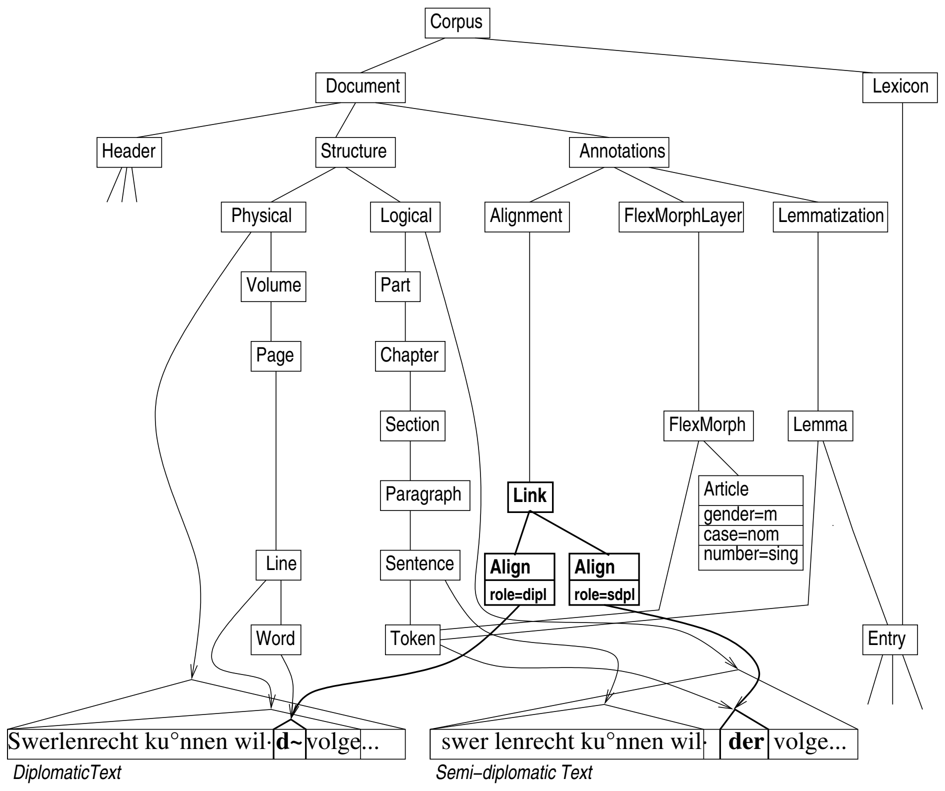


Figure 3.3: Detail of a prototypical instance of the DDD schema (from the Heidelberger Handschrift of the Sachsenspiegel, early 14th century).

The diplomatic and semi-diplomatic text layers are aligned using an Alignment element. Fig. 3.3 shows that the abbreviation “d˜” is aligned with its expansion “der” via a Link node containing two Align elements that refer to the spans to be aligned.

Part-of-speech annotation and inflectional morphology information is organized in the FlexMorphLayer. A FlexMorph element associates a token and one or more elements inheriting from FlexMorphTag (cf. Fig. 3.6). The part of speech information is represented by the tag name (e.g., Noun, Verb). Further information is stored in form of attributes (e.g., number, case).

The encoding technique for alignments is further demonstrated in Fig. 3.4 for a bilingual Latin / Old High German text (Tatian). Corresponding spans in Latin and in Old High German are referenced by the Align children of a common Link element, distinguished by their role attribute. For instance, the Align elements in Fig. 3.4 that point to the Latin text have role="lat" whereas those pointing to the Old High German text have role="goh".

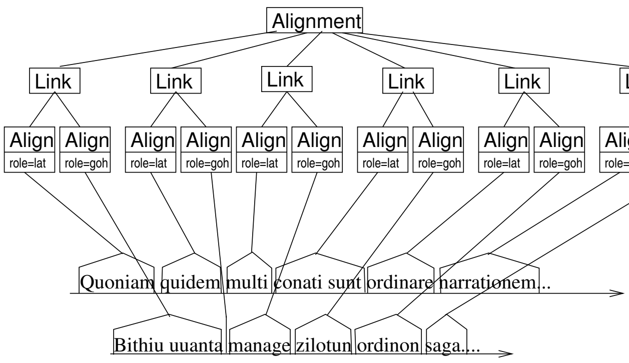

Alignment
Link
Link
Link
Link
Link
Link
Align role=lat
Align role=goh
Quoniam quidem multi conati sunt ordinare narrationem...
Bithiu uuanta manage zilotun ordinon saga...

Figure 3.4: Alignment of corresponding text spans for a bilingual Latin / Old High German text, Tatian

Instead of aligning *spans* in a text, an alignment can align (sequences of) *elements*. Fig. 3.5 shows an alternative alignment for the Tatian example from Fig. 3.4 where each Align element has one or more Token elements as children.

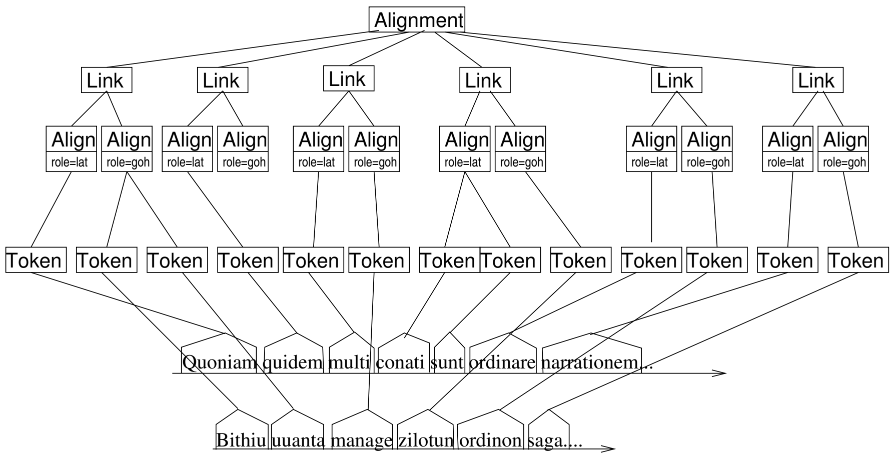

Alignment
Link
Link
Link
Link
Link
Link
Align role=lat
Align role=goh
Token
Quoniam quidem multi conati sunt ordinare narrationem...
Bithiu uuanta manage zilotun ordinon saga...

Figure 3.5: Alignment of token elements for a bilingual source text (Latin - Old High German), Tatian

The preliminary schema of the DDD corpus is shown as an UML class model in Fig. 3.6. The model is a refinement of the data model shown in Fig. 3.2. Its nodes define subclasses of Element. The schema specifies how elements such as Token can be included within several other parent elements, i.e., Sentence, Lemma, FlexMorph.

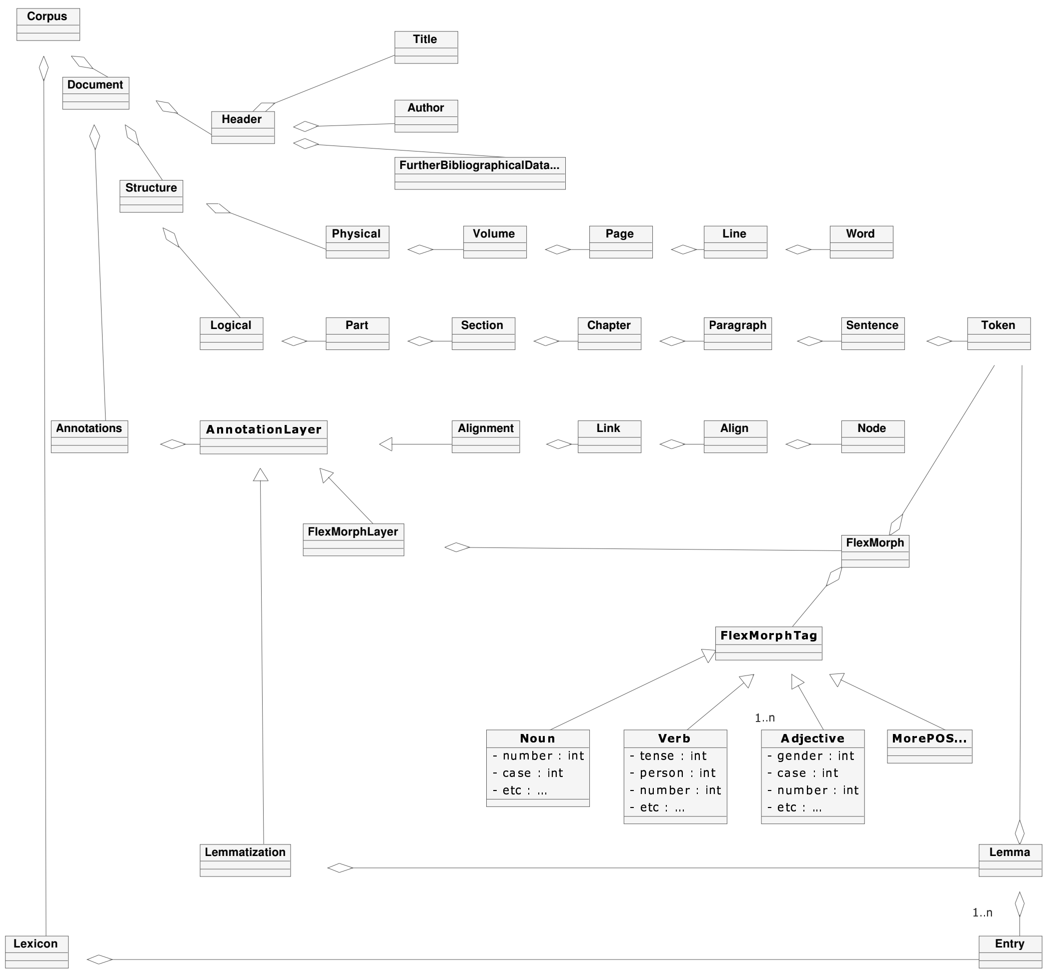


Figure 3.6: UML class model of preliminary DDD schema.

3.3 Storage

3.3.1 Exchange Format

As an external exchange format for ODAGs, we have developed the XML format gXDF. It specifies attributes that allow elements to refer to text spans. Cases of nodes having multiple parents are encoded by node references. This format is used to import ODAGs into the database where they are converted into the relational ODAG representation by resolving these node references.

3.3.2 Storing ODAGs in a Relational Database System

Methods for storing ODAGs in an RDBMS have been investigated in [Vitt, 2004]. The relational schema presented here is based on this work.

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`.

Text spans are modeled using a structured user-defined type `Span` with attributes `tid` referring to the identifier of a text layer, `left` specifying the start position within this text layer, `right` the next position after the last position, and `score` a similarity score used in inexact (fuzzy) searches. A new span from position l to position r in text layer t with score s is constructed in SQL by the expression `Span(`t`,`l`,`r`,`s`)`.

ODAG elements are stored in table

`element(id, name, span)`

The attribute `span` specifies an (optional) span associated with the element.

Since the ODAG data model is a generalization of XML, our storage concept is based on a shredded interval-based storage scheme similar to [Grust et al., 2004]. In this model, each document node is stored together with its so-called pre-order and post-order ranks. These ranks result from traversing the document tree in a certain way (depth-first pre-order or depth-first post-order) and numbering all nodes in the order of the traversal. This representation allows queries for the XPath axes to be translated into simple conditions on rank-intervals.

The interval-based approach for storing XML is generalized to support the ODAG data model. An ODAG element with multiple parents will be visited multiple times during a traversal. Hence several ranks can be attached to an element, each of which corresponds to a different visit of this element:

`rank(element, pre, post, parent)`

Attribute `pre` stores a pre-order rank of the element referenced by attribute `element` and attribute `post` stores the corresponding post-order rank. The `parent` attribute references the parent *element* from which this visit described by the rank tuple came.

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`.

3.4 Search Operators

The semantic building blocks of a query language are operators. We take the approach to define such operators as first-order logic predicates. The semantics of such an atomic predicate is specified in terms of a conceptual SQL implementation, i.e., a parameterized SQL template. A concrete implementation in a particular database system will be semantically equivalent, but may be more efficient (for instance, by taking advantage of vendor-specific features).

The meaning of a whole query can be specified then as a logic formula over these atomic predicates. This formula is translated into SQL by replacing each occurrence of an atomic predicate with an instance of its SQL template. The resulting formula of SQL queries is then transformed into a single SQL query (see Sec. 3.4.7) and sent to the database where it will be optimized further and finally executed.

The definition of a query language syntax on top of this semantic basis is subject to future work.

As discussed in Sec. 2.2.2, the results of a search are structured, i.e., each hit may be an association of several text spans or annotation elements. Hence the result set of a search is modeled as a sequence of tuples whose attributes store spans, element references, or scalar values (e.g., attribute values).

3.4.1 Character Sequences

Character sequences within a text layer can be searched by the following predicates:

Exact string match. The predicate that binds s to every span in text layer t marking an occurrence of string c is denoted string(t,c,s).

In SQL:1999 this predicate can be implemented naively using the function `POSITION(. IN .)`. Predicate string(t,c,s) finds all spans s in a given text layer t containing the given string c.

```
string(+t, +c, −s) ≡
  ⟨s⟩ ∈ (
    SELECT Span(t,p.pos,p.pos + length(c), 1)
    FROM (SELECT POSITION(c IN content) AS pos
          FROM text WHERE id=t ) p
  )
```

Note that the binding modes are indicated by attaching signs to the parameters: $+$ indicates a parameter that must be bound before calling the predicate and $-$ indicates a parameter that is bound as a result of executing the predicate.

A more efficient implementation would require a *positional* full text index.

Fuzzy string match. Predicate fuzzy$(+t, +c, -s)$ binds s to every span the content of which is similar to the string c. The similarity score is returned in `s.score`.

Standard SQL does not support proper fuzzy string matching. An Oracle-based implementation might take advantage of the `fuzzy()` search operator of Oracle Text.

Regular expression match. SQL:1999 supports regular expression matching with the `(. SIMILAR TO .)` function. However, this specification is not available in all commercial DBMS. Oracle offers the `REGEXP_INSTR()` operator instead (since version 10g). However, the linear complexity of regular expression matching is quite costly for large texts. Index support for regular expression matching is subject of research, see for instance [Cho and Rajagopalan, 2002].

3.4.2 Prefixes or Suffixes of Text Layers

A span referring to the prefix or suffix of length n of a text layer t can be retrieved using the predicates firstN(t, n, s) and lastN$(+t, +n, -s)$, respectively:

```
firstN(+t, +n, −s) ≡
  ⟨s⟩ ∈ (
    SELECT Span(t,0,LEAST(n,LENGTH(content)),1)
    FROM text
    WHERE id=t
  )
```

```
lastN(+t, +n, −s) ≡
  ⟨s⟩ ∈ (
    SELECT Span(t,
      GREATEST(0, LENGTH(content)- n),
      LENGTH(content),1)
    FROM text
    WHERE id=t
  )
```

3.4.3 Single Elements and Attributes

Elements within an annotation layer can be specified only by name. Predicate element(t, e) is satisfied iff e is an element with name t:

$\text{element}(+t, -e) \equiv$
```
  ⟨e⟩ ∈ (
    SELECT e.id
    FROM element e
    WHERE e.name= t
  )
```

Attributes are specified by their name. Predicate attribute(e, a, v) is satisfied iff the attribute named a of element e has value v:

$\text{attribute}(?e, ?a, -v) \equiv$
```
  ⟨e,a,v⟩ ∈ (
    SELECT a.element, a.name, a.value
    FROM attribute a
    WHERE 1=1
    [AND a.element=e]bound(e)
    [AND a.name= a]bound(a)
  )
```

Note the notation $?p$ for a parameter p that is optionally bound and the notation $[\ldots]_c$ for an optional SQL fragment that is only included if condition c is satisfied at the time the predicate is called.

Typically one searches for attributes within all elements with a given tag name. This can be achieved by the combining operators discussed next.

3.4.4 Spans

The span s of an element e is retrieved by

$\text{elementSpan}(+e, -s) \equiv$
```
  ⟨s⟩ ∈ (
    SELECT e.span
    FROM element e
    WHERE e.id= e
    AND e.span IS NOT NULL
  )
```

The attributes of a span can be accessed using the predicate span:

$\text{span}(-t, -l, -r, -c, +s) \equiv$
```
  ⟨s⟩ ∈ (
    SELECT s.tid, s.left, s.right, s.score
  )
```

A new span can be constructed using the second definition of span:

```
span(+t, +l, +r, +c, −s) ≡
  ⟨s⟩ ∈ (
    SELECT Span(t,l,r,c)
  )
```

The content of a span can be retrieved using content:

```
content(+s, −c) ≡
  ⟨c⟩ ∈ (
    SELECT SUBSTR(t.content, s.left, s.right - s.left)
    FROM text t
    WHERE t.id= s.tid
  )
```

The convenience predicate

$$\text{elementContent}(+e, -c) \equiv \text{elementSpan}(e, s) \wedge \text{content}(s, c)$$

returns the content of the span the element e is referring to:

```
elementContent(+e, −c) ≡
  ⟨c⟩ ∈ (
    SELECT SUBSTR(t.content,
                  e.span.left,
                  e.span.right - e.span.left)
    FROM element e, text t
    WHERE e.id= e
    AND e.span IS NOT NULL
    AND e.span.tid = t.id
  )
```

3.4.5 Spatial Predicates

In addition to the equality predicate . = . there are more general predicates on pairs of spans that test for spatial relationships within the positions of a text layer. These predicates are defined as boolean conditions to be used in a SQL `WHERE`-clause. However, every boolean condition c can be rewritten into the equivalent condition (`SELECT 1 WHERE` c) $\neq \emptyset$ on a stand-alone SQL query.

- contains(s, s') ≡
  ```
  s.tid = s'.tid AND
  s.left <= s'.left AND
  s'.right <= s.right
  ```

Span s is contained in span s'.

This predicate could be used for instance to find all word elements within a certain page element.

- prefix$(s,s') \equiv$
  ```
  s.tid = s'.tid AND
  s.left = s'.left AND
  s.right <= s'.right
  ```
  Span s is a prefix of span s'.

- suffix$(s,s') \equiv$
  ```
  s.tid = s'.tid AND
  s'.left <= s.left AND
  s.right = s'.right
  ```
  Span s is a suffix of span s'.

- overlaps$(s,s') \equiv$
  ```
  s.tid=s'.tid AND
  s'.left < s.right AND
  s.left < s'.right
  ```
  Span s overlaps with span s'.

- immediatelyPrecedes$(s,s') \equiv$
  ```
  s.tid=s'.tid AND
  s.right=s'.left
  ```
  Span s is immediately followed by span s'.

- precedes$(s,s') \equiv$
  ```
  s.tid=s'.tid AND
  s.right <= s'.left
  ```
  Span s ends before span s' starts.

- startsBefore$(s,s') \equiv$
  ```
  s.tid=s'.tid AND
  s.left < s'.left
  ```
  Span s starts before span s' starts.

Although span operators such as contained() and overlaps() can be computed by a simple comparison of the span boundaries, joins on intervals are not supported very efficiently by current RDBMS because they are designed primarily for efficient equijoins on single attributes. In [Enderle et al., 2004], an approach for supporting efficient interval-joins on top of an ORDBMS is presented.

Spans can be combined using the following predicates which are defined as SQL expressions. An SQL expression e can be turned into the stand-alone query `SELECT` e.

- concat$(+s_1, +s_2, -s) \equiv$
  ```
  s = CASE
          WHEN immediatelyPrecedes(s1,s2)
          THEN Span(s1.tid, s1.left, s2.right, 1)
        END
  ```
  Computes s as the concatenation of s_1 and s_2 or `NULL` if s_1 does not precede s_2 directly.

- intersection$(+s_1, +s_2, -s) \equiv$
  ```
  s = CASE
          WHEN overlaps(s1,s2)
          THEN Span(s1.tid,
                    GREATEST(s1.left, s2.left),
                    LEAST(s1.right, s2.right),
                    1)
        END
  ```

3.4.6 Hierarchical Navigation

Hierarchical relationships between elements are supported by the following operators.

- parent$(?e, ?e') \equiv$
  ```
  ⟨e,e'⟩ ∈ (
     SELECT r.parent, r.element
     FROM rank r
     WHERE 1=1
     [AND r.element= e' ]bound(e')
     [AND r.parent= e ]bound(e)
  )
  ```
  Element e is a parent of element e' in the context of a rank `r` of e'.

- ancestor$(?e, ?e') \equiv$
  $\langle e, e' \rangle \in ($
  ```
  SELECT a.element, d.element
  FROM rank a, rank d
  WHERE d.pre BETWEEN a.pre AND a.post
  [AND d.element= e′ ]bound(e′)
  [AND a.element= e ]bound(e)
  ```

  Element e has a rank a that is an ancestor of a rank n of element e'.

  This predicate uses the pre-/post-order rank encoding technique described for instance in [Grust et al., 2004] to avoid a costly recursive traversal of the ODAG.

Note that an element may cover the same span as some of its ancestors. Hence the contains() relation (on the spans of elements) is coarser than the ancestor() relation. The ancestor() relation is used in querying syntax trees since it formalizes the linguistic concept of *syntactic dominance* which cannot be expressed properly in terms of the contains() relation.

3.4.7 Boolean Operators

A conjunction / disjunction of conditions is expressed as a join / union on the tables specified by the constituent conditions.

A Normal Form of Atomic Predicates. All definitions of search operators given so far can be rewritten into the following normal form:

```
p(v_1,...,v_n) ≡
  ⟨v_1,...,v_n⟩ ∈ (
    SELECT e_1,...,e_n
    FROM R_1 AS r_1,...,R_k AS r_k
    WHERE C(r_1,...,r_n,v_1,...,v_n)
  )
```

Binding modes are ignored by this form. Only *input* variables are actually used in condition $C()$. For every input variable v_i, the corresponding column expression e_i is v_i.

Conjunction. A conjunction $p(u_1,\ldots,u_m,v_1,\ldots,v_n) \wedge q(v_1,\ldots,v_n,w_1,\ldots,w_o)$ with shared variables $v_1,\ldots,v_n$ where predicate p is defined as

$p(x_1,\ldots,x_{m+n}) \equiv$
  $\langle x_1,\ldots,x_{m+n} \rangle \in ($

```
    SELECT $a_1$, …, $a_{m+n}$
    FROM $R_1$ AS $r_1$, …, $R_k$ AS $r_k$
    WHERE $C(r_1, \ldots, r_n, x_1, \ldots, x_{m+n})$
  )
```

and predicate q is defined as

```
$q(y_1, \ldots, y_{n+o}) \equiv$
  $\langle y_1, \ldots, y_{n+o} \rangle \in$ (
    SELECT $b_1$, …, $b_{n+o}$
    FROM $S_1$ AS $s_1$, …, $S_l$ AS $s_l$
    WHERE $D(s_1, \ldots, s_l, y_1, \ldots, y_{n+o})$
  )
```

yields a condition of the same form which combines the two definitions, replaces the shared variables $v_1, \ldots, v_n$ by the column expressions $b_1, \ldots, b_n$ in the conditions $C()$ and $D()$ and adds join conditions $a_{m+i} = b_i$ for the column expressions of those shared variables $v_1, \ldots, v_n$ that are output variables of both predicates (i.e., $v_i \notin \{a_{m+i}, b_i\}$):

```
$p(u_1, \ldots, u_m, v_1, \ldots, v_n) \wedge q(v_1, \ldots, v_n, w_1, \ldots, w_o) \equiv$
  $\langle u_1, \ldots, u_m, v_1, \ldots, v_n, w_1, \ldots, w_o \rangle \in$ (
    SELECT $a_1$, …, $a_m$, $c_1$, …, $c_n$, $b_{n+1}$, …, $b_{n+o}$
    FROM $R_1$ AS $r_1$, …, $R_k$ AS $r_k$,
         $S_1$ AS $s_1$, …, $S_l$ AS $s_l$
    WHERE $C(r_1, \ldots, r_n, u_1, \ldots, u_m, b_1, \ldots, b_n)$
    AND   $D(s_1, \ldots, s_l, b_1, \ldots, b_n, w_1, \ldots, w_o)$
    AND   $[a_{m+1} = b_1]_{v_1 \notin \{a_{m+1}, b_1\}}$
    …
    AND   $a_{m+n} = b_n]_{v_n \notin \{a_{m+n}, b_n\}}$
  )
```

where $c_i = a_{m+i}$ if $a_{m+i} \not\equiv v_i$, $c_i = b_i$, otherwise. (Without loss of generality we assume $\{r_1, \ldots, r_k\}$ and $\{s_1, \ldots, r_l\}$ to be disjoint.)

This joint condition forms the definition of a predicate equivalent to the conjunction $p(u_1, \ldots, u_m, v_1, \ldots, v_n) \wedge q(v_1, \ldots, v_n, w_1, \ldots, w_o)$.

Disjunction. A disjunction $p(u_1, \ldots, u_m, v_1, \ldots, v_n) \vee q(v_1, \ldots, v_n, w_1, \ldots, w_o)$ with shared variables $v_1, \ldots, v_n$ where predicates p and q are defined as above yields predicate based on a SQL `UNION` query:

```
$p(u_1, \ldots, u_m, v_1, \ldots, v_n) \vee q(v_1, \ldots, v_n, w_1, \ldots, w_o) \equiv$
  $\langle u_1, \ldots, u_m, v_1, \ldots, v_n, w_1, \ldots, w_o \rangle \in$ (
    (
      SELECT $a_1$, …, $a_{m+n}$, NULL, …, NULL
      FROM $R_1$ AS $r_1$, …, $R_k$ AS $r_k$
      WHERE $C(r_1,\ldots,r_n,u_1,\ldots,u_m,v_1,\ldots,v_n)$
    )
      UNION
    (
      SELECT NULL, ..., NULL, $b_1,\ldots,b_{n+o}$
      FROM $S_1$ AS $s_1,\ldots,S_l$ AS $s_l$
      WHERE $D(s_1,\ldots,s_l,v_1,\ldots,v_n,w_1,\ldots,w_o)$
    )
  )
```

Using structural induction, we conclude:

Corollary 3.4.1 Every query consisting of conjunctions and disjunctions of atomic predicates in normal form as defined above can be rewritten as an SQL query which is an `UNION` of `SELECT` statements.

Negation. Let p be a predicate in normal form. Without loss of generality, we assume that all parameters are input parameters (mode $+$). This can be achieved by omitting all parameters with other modes. Then the negation $\bar{p}$ is defined as:

```
$\bar{p}(v_1,\ldots,v_n) \equiv$
  $\langle v_1,\ldots,v_n\rangle \in$ (
    SELECT $v_1,\ldots,v_n$
    WHERE NOT EXISTS(
      SELECT 1
      FROM $R_1$ AS $r_1,\ldots,R_k$ AS $r_k$
      WHERE $C(r_1,\ldots,r_n,v_1,\ldots,v_n)$
    )
  )
```

This is again a predicate in normal form.
Using structural induction, we conclude:

Corollary 3.4.2 Every query consisting of conjunctions, disjunctions, and negations of atomic predicates in normal form as defined above can be rewritten as an SQL query which is an `UNION` of `SELECT` statements.

Examples. For instance, query $Q_1(t,a,e,v)$ which retrieves the value v of attribute a for every element e named t can be specified as

$$Q_1(t,a,e,v) \quad\equiv\quad \text{element}(t,e) \wedge \text{attribute}(e,a,v)$$

which expands to:

```
$Q_1(t,a,e,v) \equiv$
  $\langle t,e \rangle \in ($
    SELECT e.name, e.id
    FROM element e
    WHERE e.name= $t$
  $) \wedge \langle e,a,v \rangle \in ($
    SELECT a.element, a.name, a.value
    FROM attribute a
    WHERE 1=1
    AND a.element=$e$
    AND a.name= $a$
  $)$
```

This conjunction translates into the following SQL query:

```
$Q_1(t,a,e,v) \equiv$
  $\langle t,a,e,v \rangle \in ($
    SELECT e.name, e.id, a.name, a.value
    FROM element e, attribute a
    WHERE e.name= $t$
    AND e.id= a.element
    AND a.name= $a$
  $)$
```

To find all word elements e whose content equals the string "*lenrecht*" in text layer t one can use the query Q_2 defined as

$$\begin{aligned} Q_2(t,e) \quad \equiv \exists s: \ & \text{string}(t, \texttt{'lenrecht'}, s) \wedge \\ & \text{element}(\texttt{'word'}, e) \wedge \text{elementSpan}(e,s) \end{aligned}$$

which expands to:

```
$Q_2(t,e) \equiv$
  $\langle s \rangle \in ($
    SELECT Span($t$,p.pos,p.pos + 8, 1)
    FROM
      (SELECT
         POSITION('lenrecht' IN content) AS pos
       FROM text WHERE id=$t$ ) p
  $) \wedge \langle e \rangle \in ($
    SELECT e.id
    FROM element e
    WHERE e.name= 'word'
  $) \wedge \langle s \rangle \in ($
    SELECT e.span
    FROM element e
    WHERE e.id= e
  )
```

This formula can be rewritten into a membership condition on a single SQL query which can be simplified further into the following definition:

```
Q2(t,e) ≡
  ⟨e⟩ ∈ (
    SELECT e.id
    FROM element e
    WHERE e.name= 'word'
    AND e.span IN (
      SELECT Span(t,p.pos,p.pos + 8, 1)
      FROM (
        SELECT POSITION('lenrecht' IN text.content) AS pos
        FROM text WHERE id= t) p
    )
  )
```

3.4.8 Sequence Operators

There are many linguistic queries where sequences of elements must be matched. For instance, one might want to find all sequences of an article followed by one or more adjectives and finally a noun. This could be specified using a regular expression such as Article Adjective$^+$ Noun. When searching e.g., syntax trees, one may want to match sequences of elements that are not necessarily siblings, but whose spans are adjacent. Hence sequence operators for matching and combining *spans* are needed. Sequence operators are defined here as second-order logic predicates which combine zero or more sequence operators. At compile-time these definitions can be expanded into (recursive) first-order logic predicates. A sequence operator takes a span s as first run-time argument and returns as its second argument a span that is the concatenation of s and a span s' matched by the sequence operator.

The trivial sequence operator just returns the input span:

$\text{empty}(+s, ?s') \equiv s = s'$

The most basic sequence operator is the *concatenation* of two spans. Predicate $\text{concat}_{p,q}(s, s'')$ is satisfied if and only if span s'' is the concatenation of s with a match for predicate p and a match for predicate q:

$$\text{concat}_{p,q}(+s, ?s'') \equiv \exists s' : p(s, s') \wedge q(s', s'')$$

An alternative $e|f$ in a regular expression can be expressed by a disjunction of two predicates p, q that implement e and f, respectively:

$$\text{alt}_{p,q}(+s, ?s') \equiv p(s,s') \vee q(s,s')$$

To offer the full expressiveness of regular expressions over spans, the Kleene star operator must be supported. The regular expression e^* can be expressed as a recursive predicate that is parameterized with a predicate p implementing the regular expression e. This predicate p is used in the definition of predicate star_p for detecting subsequent matches of e:

$$\text{star}_p(+s, ?s'') \equiv \text{empty}(s,s'') \vee (p(s,s') \wedge \text{star}_p(s',s''))$$

In SQL the star operator translates into a recursive self join, a rather costly operation (if it is supported by the underlying database system at all). A more efficient method to compute this may be to sort a table by the left border of the span attribute and then use an external or stored procedure to sequentially aggregate consecutive tuples having adjacent spans. The span attributes are aggregated to the concatenation of all contributing spans, all other attributes can be combined using the usually available SQL aggregation operators. As an alternative, one could just add the aggregated span as a new attribute to each contributing tuple. Note, that every tuple may contribute to multiple aggregations and would have to be replicated in this case.

3.4.9 Advanced Operators

Further query operators whose definition is postponed to future work are:

- alignment operators for projecting spans across aligned text layers
- operators for selecting / combining text variants
- statistical aggregation operators for counting, averaging, etc.
- operators for computing collocations[1]

3.5 Query Examples

3.5.1 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)

[1]see www.collocations.de

with conditions on the lemma annotation (lemma name n equals "sagen") and the inflectional morphology f. Fig. 3.7 shows an example of an ODAG subgraph matching this query.

$$\begin{aligned} Q_a(s) \equiv\ & \text{element}(\texttt{'Sentence'}, s) \wedge \text{element}(\texttt{'Token'}, t) \wedge \text{ancestor}(s,t) \wedge \\ & \text{element}(\texttt{'Lemma'}, l) \wedge \text{parent}(l,t) \wedge \\ & \text{element}(\texttt{'Entry'}, e) \wedge \text{parent}(l,e) \wedge \\ & \text{element}(\texttt{'LemmaName'}, n) \wedge \text{parent}(e,n) \wedge \\ & \text{elementSpan}(n, s_n) \wedge \text{string}(s_n.\texttt{tid}, \texttt{'sagen'}, s_n) \wedge \\ & \text{element}(\texttt{'FlexMorph'}, f) \wedge \text{parent}(f,t) \wedge \\ & \text{element}(\texttt{'Verb'}, v) \wedge \text{parent}(f,v) \wedge \\ & \text{attribute}(v, \texttt{'person'}, 2) \wedge \text{attribute}(v, \texttt{'number'}, \texttt{'sing'}) \end{aligned}$$

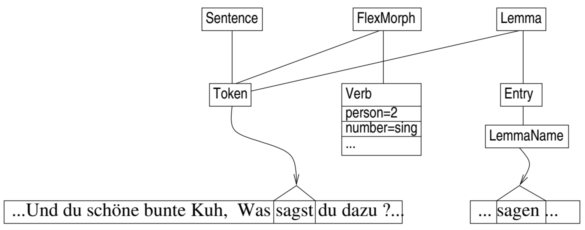


Figure 3.7: A match for query Q_a in a hypothetical DDD edition of [Grimm and Grimm, 1812].

3.5.2 Querying aligned texts

How is "pulcher" (lat.) translated into Old High German? An alignment in the representation shown in Fig. 3.4 is assumed. 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.

$$\begin{aligned} Q_b(s_g) \equiv\ & \text{element}(\texttt{'Token'}, t) \wedge \text{elementSpan}(t, s_t) \wedge \\ & \text{string}(s_n.\texttt{tid}, \texttt{'pulcher'}, s_n) \wedge \\ & \text{element}(\texttt{'Align'}, a_l) \wedge \text{attribute}(a_l, \texttt{'role'}, \texttt{'lat'}) \wedge \\ & \text{elementSpan}(a_l, s_l) \wedge \text{contains}(s_l, s_t) \wedge \\ & \text{parent}(l, a_l) \wedge \text{element}(\texttt{'Link'}, l) \wedge \\ & \text{parent}(l, a_g) \wedge \text{element}(\texttt{'Align'}, a_g) \wedge \end{aligned}$$

$$\text{attribute}(a_g, '\texttt{role}', '\texttt{goh}') \land$$
$$\text{elementSpan}(a_g, s_g)$$

3.5.3 Querying Linguistic Trees

The following sample queries are taken from [Bird et al., 2005]. 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*.

$$\begin{aligned} Q_1(np, v) \equiv\ & \text{element}('\texttt{V}', v) \land \text{element}('\texttt{NP}', np) \land \\ & \text{elementSpan}(v, s_v) \land \text{elementSpan}(np, s_{np}) \land \\ & \text{immediatelyPrecedes}(s_v, s_{np}) \end{aligned}$$

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

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

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

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

3.6 Transformation

To present the result of a query to the user they must be transformed into a format such as (X)HTML or PDF. We advocate a combined approach where the necessary data is exported from the database in a generic XML format and is then transformed using an XSLT stylesheet that is compiled or parameterized from the user query.

Chapter 4

Related Work

4.1 Query Language

Numerous linguistic query tools have been developed in the last years. While some tools just provide a graphical user interface for entering search terms and conditions, others require the user to formulate queries in a specialized query language. Here we discuss the linguistic query languages CQP, Tiger, and LPath as well as the general-purpose XML query languages XPath and XQuery as a possible basis for building a query language for DDD. Other linguistic query languages that are not discussed here but should also be considered include for instance tgrep[1], CorpusSearch[2], the NITE Query Language [Evert and Voormann, 2002], and Emu[3][Cassidy and Harrington, 2001]. Corpus query tools without query language are for instance SARA / Xaira[4], or COSMAS / COSMAS II [5].

4.1.1 Corpus Query Processor (CQP)

As part of the Corpus Work Bench, CQP [Christ, 1994] is well-known in the corpus linguistics community.

- Simple queries for literal strings or regular expressions can be expressed without syntactic overhead (e.g., `"example"`, `"walk(ed)?"`).
- Positional annotations are represented as token attributes (e.g., `[pos="NN"]`). Queries for literals or regular expressions (see above) are actually syntac-

[1]`http://www.ldc.upenn.edu/ldc/online/treebank/`
[2]`http://www.ling.upenn.edu/~dringe/CorpStuff/Manual/Contents.html`
[3]`http://emu.sourceforge.net/`
[4]`http://www.oucs.ox.ac.uk/rts/xaira/`
[5]`http://www.ids-mannheim.de/cosmas2/`

tic sugar for conditions on the `word` attribute (e.g., `[word="example"]` and `[word="walk(ed)?"]`).

- Conditions on tokens can be combined to specify sequences (e.g., `"give" []{0,3} "up"` specifies a sequence of the word "give", followed by zero to three arbitrary tokens, followed by the word "up").
- Since CQP is token-based, it cannot be used to return sequences whose boundaries do not coincide with token boundaries.
- It provides rudimentary support for restricting search to hits co-occuring within certain XML elements (e.g., `"give" []* "up" within s` restricts the search to matches within the same `s` element), but does not support more complex structural conditions (such as path expressions).
- Sentence-aligned corpora are supported (e.g., `:french "neuf" :english != "new"` finds all occurrences of "neuf" in the `french` corpus within a sentence being aligned with a sentence in the `english` corpus that does not contain the word "new"). However, conditions on characterwise alignments cannot be expressed.

The DDD query language should inherit from CQP the easy specification of searches for literals, regular expressions, and token sequences.

4.1.2 TigerSearch

TigerSearch [Brants et al., 2002, Lezius, 2002] has been designed for querying syntax trees in the Tiger tree-bank. It uses a syntax similar to CQP for specifying token attributes (e.g., `[word="Abend" & pos="NN"]`) and extends it to non-terminal nodes (e.g., `[cat="NP"]` for searching for a noun phrase). Variables can be assigned to nodes (e.g., `#np:[cat="NP"]`). To express relationships between nodes, it introduces several operators for direct dominance (parent-child relationship, e.g., `#n1 > #n2`), labeled dominance (allows to specify the edge label; e.g., `#n1 >HD #n3`), dominance (ancestor-descendent relationship, e.g., `#n1 >* #n3`), textual precedence (e.g., direct `#n1 . #n2` or with arbitrary distance `#n1 .* #n2`) and several other linguistically motivated operators (e.g., left corner dominance).

4.1.3 XPath / XQuery

XPath provides an intuitive and standardized way to express navigational queries. However, the schema of the corpus must be known in detail even for basic queries such as searching for occurrences of a string. This also holds for XQuery which uses XPath expressions to specify paths.

XQuery 1.0 and XPath 2.0 Full-Text [6] adds information retrieval operators to XPath expressions which can be used in XQuery or XSLT. The boolean operator NodeSequence ftcontains FTSelection is satisfied if the NodeSequence contains at least one node whose text content matches the FTSelection. The latter is essentially a boolean expression over string literals (e.g., `"dog" || "cat"`) with additional options for stemming, regular expressions etc. By wrapping a `ftcontains` condition with a `ft:score()` function, one can retrieve a score for a match instead of a boolean value. The specification does not allow to retrieve the positions of matching substrings. This is a severe limitation in the linguistic context. However, this emerging standard should be leveraged as much as possible in defining the DDD query language.

LPath[Bird et al., 2005] is an extension of XPath for querying tree banks. It introduces the new navigation axes *immediate-following/preceding* and to access nodes whose text span immediately follows the text span of the current node. Subtree scoping allows to restrict all navigations within a subquery to the tree rooted in the current node. This is important since several axes leave this subtree by default. Finally, there are operators ˆ and $ for edge alignment that can be used to find nonterminals whose left/right border coincides with the left/right border of a surrounding nonterminal. We consider LPath as an important step towards a linguistic query language based on XML standards. For our purposes, XPath needs further extensions to

- cope with elements having multiple parents
- access the span an element refers to
- find elements referring to a span
- project a span on one text layer to a span on an aligned text layer
- test for spatial relationships of spans (overlap, inclusion, precedence, etc.)
- specify regular expressions on node sequences

[6] `http://www.w3.org/TR/xquery-full-text/`

XQuery. In [Cassidy, 2002] several use cases for XQuery queries over a multi-modal speech corpus are discussed. XPath expressions in XQuery are found to be useful to express hierarchical queries while sequential constraints cannot be expressed as easily as in the query language AGQL[Bird et al., 2000] proposed for annotation graphs. A use case is presented where a correlated subquery with an ∀-quantifier (`EVERY ... IN ... SATISFIES`) is needed. This cannot be expressed in XPath.

While we understand that a general purpose query language such as XQuery provides the most expressive basis for a linguistic query language, we feel that first XPath should be extended to yield a basic linguistic query language that could be used later as a component within a linguistic extension of XQuery.

4.2 Linguistic Data Models

A data model based on ordered, acyclic graphs (ODAGs) has been proposed in [Dipper et al., 2004] for DDD. Our approach has been inspired by prior work in the field of multi-modal corpora, namely the NITE Object Model. The data model of the speech database Emu is also similar to our ODAG data model. These and other data models for linguistic corpora are presented here.

Annotation Graphs (AG) [Bird and Liberman, 2001] and NITE Object Model (NOM) [Carletta et al., 2003] There are two popular data models for multi-modal corpora: the annotation graph (AG) model [Bird and Liberman, 2001] and ordered directed acyclic graphs (ODAGs), such as the NITE object model (NOM) [Carletta et al., 2003]. Annotation graphs model annotations as arcs that connect time points on the time axis of a signal. Annotation graphs can be stored easily in relational databases and searched efficiently by translating queries into SQL. However, the AG model has some shortcomings. For instance, parent-child relationships cannot be represented in AGs without extending the data model with special child/parent arcs [Teich et al., 2001]. Without this extension, the dominance relation between a non-branching node and its only child is not encoded. Meta-annotations or alignments cannot be represented directly but need to be expressed by introducing equivalence classes (i.e., annotations are linked by assigning them identical attribute values).

The ODAG-based NOM does not share these limitations. Annotations are represented by nodes. Annotation values are stored in form of node attributes. The domination relation between nodes is modeled explicitly by parent-child relationships. Each node may refer to a span of the underlying text. In this case, the child nodes must refer to non-overlapping text spans contained in the span of their parent node. The order of child nodes must correspond to the order of their spans in the underlying text.

Multi-colored Trees. In [Jagadish et al., 2004] the *multi-colored tree* (MCT) model, a new logical data model based on the XML data model [Fernandez and Robie (Eds), 2003] is introduced. MCT allows nodes to be shared by multiple document trees distinguished by colors. A shared node may have different children and attributes in each tree. Hence the same data elements can be organized in different hierarchies.

The MCT model is motivated with modeling considerations: it is convenient to organize document elements in a single hierarchy. However, to avoid redundancy, hierarchies have to be broken up by introducing references that are less convenient to handle. By supporting multiple XML trees over the same data, MCT avoids this tradeoff.

MCT could be used to represent conflicting hierarchies in linguistically annotated texts. Each hierarchy would be labeled with a different color. Correspondences between nodes of different hierarchies would be expressed in another color.

Directed acyclic graphs can be encoded in MCT by introducing different colours for different parents of a node. However, the number of colors depends on the concrete graph and may grow exponentially in the number of nodes having multiple parents. Moreover, to retrieve all parents of a node, one would have to enumerate all colors, an operation that is not supported by the multi-colored XQuery extension described in [Jagadish et al., 2004].

XTE (eXternal Text Encoding) [Simonis, 2004] is an XML format that supports the storage of multiple parallel text layers and multiple conflicting annotation hierarchies over these text layers within a single file. This is very similar to the gXDF XML format, an exchange format for the ODAG model. Text layers are stored separate from the annotations. Each annotation element may refer to a span in a text layer. The XTE format is the storage format of the LanguageExplorer[7] and LanguageAnalyzer tools for presentation and editing of parallel texts.

Emu [Cassidy and Harrington, 2001] is a speech database system. It organizes a set of utterances (e.g., spoken sentences) each of which consists of a set of annotation levels. Each annotation level stores a set of tokens that may carry timing information. Tokens can be associated with each other by sequential, hierarchical, and user-defined relations. The sequential relation defines a partial ordering of tokens that must be consistent with their timing information. Hierarchical relations (linguistic dominance) associate a parent token with an ordered

[7] `http://www.language-explorer.org/`

sequence of child nodes. They must be acyclic and can exist both within or across levels but must not induce ambiguities in the sequential ordering.

Chapter 5

Conclusions and Future Work

In this report, we have investigated methods for querying multi-layered richly-annotated linguistic corpora such as the planned DDD corpus. We have identified requirements, have defined basic query operators as first-order logic predicates, and have provided a conceptual implementation in SQL.

These query operators provide a basis for defining a powerful linguistic query language. The constructs of this language will be defined in terms of logic formulas over the defined query operators. The query language should be both intuitive for the intended user community and easy to learn for users familiar with existing standards such as XPath and XQuery.

As future work we plan to extend the DDD query language with operators for handling alignments and text variants, for statistical analysis and for collocation analysis.

An optimizing translation of queries represented as logic formulas over the defined query operators into efficient SQL is needed. The resulting SQL queries should take advantage of existing functionality for full-text retrieval and management of XML. In addition, special indexing techniques for substring matching, regular expressions, and interval joins need to be reviewed more closely.

To assess the feasibility and scalability of the methods proposed in this work, performance evaluations on prototype corpora are planned.

Other topics that need to be investigated in future studies are the transformation between the internal ODAG format and external XML and non-XML formats and in particular the online-presentation of texts. This will require not only research on format conversion techniques, but will also raise ergonomic issues.

Bibliography

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

[Bird et al., 2005] Bird, S., Chen, Y., Davidson, S., Leea, H., and Zheng, Y. (2005). Extending xpath to support linguistic queries. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, Long Beach, California.

[Bird and Liberman, 2001] Bird, S. and Liberman, M. (2001). A formal framework for linguistic annotation. *Speech Communication*, 33(1,2):23–60. `http://arxiv.org/abs/cs/0010033`.

[Brants et al., 2002] Brants, S., Dipper, S., Hansen, S., Lezius, W., and Smith, G. (2002). The TIGER treebank. In *Proceedings of the Workshop on Treebanks and Linguistic Theories, September 20-21*, Sozopol, Bulgaria.

[Carletta et al., 2003] Carletta, J., Kilgour, J., O'Donnell, T., Evert, S., and Voormann, H. (2003). The NITE object model library for handling structured linguistic annotation on multimodal data sets. In *Proceedings of the EACL Workshop on Language Technology and the Semantic Web (3rd Workshop on NLP and XML, NLPXML-2003)*.

[Cassidy, 2002] Cassidy, S. (2002). Xquery as an annotation query language: a use case analysis. In *Proceedings of LREC 2002*.

[Cassidy and Harrington, 2001] Cassidy, S. and Harrington, J. (2001). Multi-level annotation in the Emu speech database management system. *Speech Communication*, 33:61–77.

[Cho and Rajagopalan, 2002] Cho, J. and Rajagopalan, S. (2002). A fast regular expression indexing engine. In *ICDE'02*, pages 419–.

[Christ, 1994] Christ, O. (1994). A modular and flexible architecture for an integrated corpus query system. In *COMPLEX'94*, Budapest.

[Dipper et al., 2004] Dipper, S., Faulstich, L. C., Leser, U., and Lüdeling, A. (2004). Challenges in modelling a richly annotated diachronic corpus of german. In *Workshop on XML-based richly annotated corpora*, Lisbon, Portugal.

[Enderle et al., 2004] Enderle, J., Hampel, M., and Seidl, T. (2004). Joining interval data in relational databases. In *SIGMOD*, pages 683–694.

[Evert and Voormann, 2002] Evert, S. and Voormann, H. (2002). Nite query language version 2.0. `http://www.ltg.ed.ac.uk/NITE/documents/NiteQL.v2.0.pdf`.

[Fernandez and Robie (Eds), 2003] Fernandez, M. and Robie (Eds), J. (2003). "XQuery 1.0 and XPath 2.0 Data Model". W3C Working Draft. `http://www.w3.org/TR/2003/WD-xpath-datamodel-20031112/`.

[Grimm and Grimm, 1812] Grimm, J. and Grimm, W. (1812). *Kinder- und Hausmärchen*, chapter 169: Das Waldhaus. Berlin: Realschulbuchhandl.

[Grust et al., 2004] Grust, T., Keulen, M. V., and Teubner, J. (2004). Accelerating XPath evaluation in any RDBMS. *ACM Transactions on Database Systems*, 29(1):91–131.

[Ide et al., 2000] Ide, N., Bonhomme, P., and Romary, L. (2000). XCES: An XML-based standard for linguistic corpora. In *Proceedings of the Second Language Resources and Evaluation Conference (LREC)*, pages 825–830.

[Jagadish et al., 2004] Jagadish, H. V., Lakshmanan, L. V. S., Scannapieco, M., Srivastava, D., and Wiwatwattana, N. (2004). Colorful xml: One hierarchy isn't enough. In *SIGMOD Conference*, pages 251–262.

[Kroymann et al., 2004] Kroymann, E., Thiebes, S., Lüdeling, A., and Leser, U. (2004). Eine vergleichende analyse von historischen und diachronen digitalen korpora. Technischer Bericht 174, Institut für Informatik der Humboldt Universität zu Berlin.

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

[Lüdeling et al., 2005] Lüdeling, A., Poschenrieder, T., and Faulstich, L. (2005). DeutschDiachronDigital, ein diachrones Korpus des Deutschen. *Jahrbuch für Computerphilologie*. In Print.

[Simonis, 2004] Simonis, V. (2004). *A framework for processing and presenting parallel text corpora.* PhD thesis, Universität Tübingen.

[Sinclair, 1996] Sinclair, J. (1996). Eagles. preliminary recommendations on corpus typology. `http://www.ilc.cnr.it/EAGLES96/corpustyp/corpustyp.html`.

[Sperberg-McQueen and Burnard, 2001] Sperberg-McQueen, C. M. and Burnard, L., editors (2001). *Guidelines for Text Encoding and Interchange*, chapter 31: Multiple Hierarchies. Text Encoding Initiative. `http://www.tei-c.org/P4X/NH.html`.

[Sperberg-McQueen and Burnard, 2002] Sperberg-McQueen, C. M. and Burnard, L., editors (2002). *Guidelines for Text Encoding and Interchange*, chapter 5: The TEI Header. TEI Consortium. `http://www.tei-c.org/P4X/HD.html`.

[Teich et al., 2001] Teich, E., Hansen, S., and Fankhauser, P. (2001). Representing and querying multi-layer corpora. In *Proceedings of the IRCS Workshop on Linguistic Databases*, pages 228–237, University of Pennsylvania, Philadelphia.

[Vitt, 2004] Vitt, T. (2004). Speicherung linguistischer korpora in datenbanken. Studienarbeit, Institut für Informatik, Humboldt Universität zu Berlin. `http://www.informatik.hu-berlin.de/Forschung_Lehre/wbi /research/stud_arbeiten/finished/2004/vitt_041114.pdf`.