Using Information Retrieval Technology for a Corpus Analysis Platform

Carsten Schnober
Institut für Deutsche Sprache
R5 6-13
D-68161 Mannheim
Germany
schnober@ids-mannheim.de

Abstract

This paper describes a practical approach to use the information retrieval engine Lucene for the corpus analysis platform KorAP, currently being developed at the *Institut für Deutsche Sprache* (IDS Mannheim). It presents a method to use Lucene's indexing technique and to exploit it for linguistically annotated data, allowing full flexibility to handle multiple annotation layers. It uses multiple indexes and MapReduce techniques in order to keep KorAP scalable.

1 Introduction

In this paper, the Lucene information retrieval (IR) framework[1] is taken out of its native compound, and ported to the field of corpus linguistics, to investigate its applicability as an engine for KorAP[2] (*Korpusanalyseplattform der nächsten Generation*, "Next Generation Corpus Analysis Platform") that is conducted at the *Institut für Deutsche Sprache* (IDS Mannheim). The aim of this project is to develop a modern, state-of-the-art corpus-analysis platform, capable of handling very large corpora and opening the perspectives for innovative linguistic research (Bański et al., 2012).

The KorAP engine is designed to be scalable and flexible enough to store and analyse the fast-growing amounts of linguistic resources that have become available to corpus linguists. The DeReKo corpus (*Deutsches Referenzkorpus*, "German Reference Corpus") serves as an immediate use case, being the largest German text corpus in the world with 5.4 billion words and three concurrent annotation layers (IDS, 2011), currently accessible through Cosmas II (Bodmer, 2005). The corpus is expected to keep growing as rapidly as it has in the past two decades of its existence in which DeReKo at least doubled its size every five years[3]. Although the actual growth cannot be extrapolated from such figures, KorAP aims to be ready for corpora with 50 billion tokens. However, the KorAP platform is not designed exclusively for DeReko but to process any corpora, independent of size, writing system, language or other specific characteristics.

1.1 Linguistic Research vs. Information Retrieval

In IR, the primary goal is to find documents containing the information the user needs. In text documents, the surface text is the vehicle that provides and often hides this information, possibly expressed in many different ways. From an IR point of view, "language is an obstacle on the way to resolving a problem" while in corpus linguistics, the language is the research object (Perkuhn et al., 2012, p. 19). A linguistic search may query complex data structures and relationships, such as multiple metrics and levels while the user demands can pose challenging inquiries (e.g. relations, quantifiers, regular expressions (Bański et al., 2012).

[1]Lucene homepage: http://lucene.apache.org
[2]KorAP: http://korap.ids-mannheim.de
[3]DeReKo outlines (in German): http://www.ids-mannheim.de/kl/projekte/korpora/archiv.html#Umfang

An ideal IR engine would thus be able to fully interpret language and to abstract its meaning. For that purpose, IR techniques typically aim to remove elements that do not bear much meaning, e.g. function words and morphological affixes. Linguistic researchers, on the other hand, are typically interested in finding linguistic phenomena of all kinds, often involving tokens that seem semantically less relevant.

1.2 Lucene

Lucene is an open-source framework for searching large amounts of text and is considered to be the most widely used information retrieval library (McCandless et al., 2010, p. 3). It essentially provides a software library for creating inverted indexes (Section 3).

The Lucene project also provides a ready-to-run search application, Solr, that implements text search with typical information retrieval functionality. However, Solr is neither designed for a linguistic application like KorAP nor easily adaptable for that purpose.

1.3 KorAP Outlines

The KorAP project aims to build a generic solution to numerous problems that have arisen with the increasing size of corpora that are subject to linguistic research. An essential specification for the KorAP engine is that it must comply with scientific requirements and therefore, its results must be falsifiable and traceable, making KorAP a scientific tool while rendering insufficient substitutes like Google Search unnecessary; cf. Kilgarriff (2007).

On the other hand, KorAP does not want to re-invent the wheel and there are numerous projects tackling similar problems, approaching from both the linguistic and the computational side. In that respect, one part of KorAP development is to investigate existing solutions and finding trade-offs between re-use and new development. Other projects taken into account for (partial) adaptation for KorAP include Annis (Zeldes et al., 2009), DDC/DWDS[4] (Sokirko, 2003), and Poliqarp (Janus and Przepiórkowski, 2007).

[4] *Das Wörterbuch der Deutschen Sprache* ("The German Language Dictionary"): `http://retro.dwds.de/`

Another central goal for KorAP is to reach a level of generalisation that makes the platform flexible enough to process any kind of data, not only text but also multimodal resources like recorded and transcribed speech; the DGD speech corpora (*Datenbank Gesprochenes Deutsch*, "German Speech Database") (Fiehler and Wagener, 2005) will serve as a use case. Neither segmentations nor annotations are constrained by the KorAP engine and it should be flexible enough to be prepared for annotation types that might not even be thought of today.

That implies that KorAP does not predefine the scale of a token; instead, any sequence of characters can be defined as the atomic building stones to which other elements can refer. KorAP reads token boundaries from external files and builds an index based on these, but does not tokenize on its own (see Section 3). Annotation tools can freely assign tags or labels to character sequences without being limited by a pre-defined tokenization algorithm or another segmentation logic.

In order to allow adding annotations dynamically at any point and independently of existing indexes, KorAP uses standoff annotations (Thompson and McKelvie, 1997) to allow a clear separation amongst annotations and between annotations and the primary text. This allows for multi-level annotations of different types, including discontinuous spans and structures with internal references like trees and dependency grammars.

2 Previous Work

Lucene, as a fast and well-established text indexing engine, has been applied for linguistic research several times in the past years. Despite the differences between IR and corpus linguistics, there is a close relation. "The ability to interrogate large collections of parsed text [...] opens the way to a new kind of information retrieval (IR) that is sensitive to syntactic information, permitting users to do more focussed search" (Ghodke and Bird, 2010).

Lucene competes with relational and XML database systems for solving the problem of efficiently querying linguistically annotated texts. "Many existing systems load the entire corpus into memory and check a user-supplied query against every tree. Others avoid the memory limitation, and use relational or XML database systems. Although these have built-in support for indexes, they do not scale up either" (Ghodke and Bird, 2010; Ghodke and Bird, 2008; Zhang et al., 2001). However, there do exist approaches in which relational database management systems (RDBMS) are applied in large scale corpus indexing applications (Schneider, 2012).

The approach presented by Ghodke and Bird (2010) is not directly transferable to KorAP because it does not focus on the same abstraction level. While KorAP strictly uses standoff annotations, Ghodke and Bird (2010) store annotations inline, attached to the primary text. Also, they create Lucene documents on the base of single sentences, which makes it elegantly easy to search for multiple words occurring in a single sentence and reduces the search space radically, but at the same time gives up information about how sentences are related to each other.

Even though this approach is certainly useful for many real-world use cases in corpus-driven linguistic research, assuming the sentence to be a naturally self-sufficient linguistic unit, this forms a restriction that, once accepted, could not be abandoned when more flexibility is needed. For instance, a query might search for co-references or repetitive utterances in subsequent sentences; especially in discourse analysis, the sentence is not always sufficient as highest research unit; cf. Volk (2002) and Sinclair (2004).

3 Inverted Indexes

Inverted indexes (Brin and Page, 1998; Knuth, 1997, Chapter 6.5) are a technique to provide fast querying for large text documents. A naïve algorithm would iterate over the full text to find occurrences of a search term; an inverted index reduces the search space by creating a dictionary that lists all the distinct terms contained in a collection of documents and all the occurrences of each term: "This is the inverse of the natural relationship, in which documents list terms" (Lucene, 2012).

In order to create an inverted index, a precise definition of a term has to be specified, corresponding to a token without implying any further (linguistic) meaning. Such tokens are the minimum units to work with because an inverted index provides direct access only to those terms that are listed in its term dictionary.

Lucene stores the term dictionary in a file that lists the distinct tokens in alphabetic order. Apart from saving disk space, this allows using common prefixes to perform Wildcard searches, for instance, efficiently. In order to make random access to the dictionary fast, an additional term info index file stores a copy of every n-th entry from the term dictionary with a delta value that defines "the difference between the position of this term's entry [...] and the position of the previous term's entry". The term index is designed to reside entirely in memory (Lucene, 2012).

A Lucene index can be segmented into multiple parts; they are listed in a dedicated file in the index directory, but fully independent of each other. A new segment is added during indexing when the previous one's size reaches a configurable threshold. New segments can be added at any later point, for instance when new documents are indexed, and multiple segments can be merged.

4 Linguistic Research with Lucene

4.1 A Lucene Analyzer for KorAP Data

The Lucene indexing process happens in three phases: read the text, analyse it, build an index. The first step may simply be reading plain text files, but can as well include text extraction from various formats like XML files, PDFs or other document types. In the second phase, the text is split into tokens that form the key terms in the inverted index.

In IR, filters are typically applied in the text analysis phase that aim to reduce the number of distinct tokens, for instance by stopword filtering and stemming. However, this is not obligatory and the analysis process is fully customisable. Lucene provides various built-in analyzers, but the choice can be broadened by own developments. After the text has been processed, the index is finally built on the basis of the tokens produced during analysis.

KorAP corpora are stored in XML files[5] where each document comprises one directory, each

[5]See `http://korap.ids-mannheim.de/2012/03/data-set-released/` for a sample KorAP XML data set.

containing a file for the text (Figure 1) and one for the metadata (Figure 2). The annotations are organised in so-called foundries, realised as dedicated directories in which one or multiple annotation levels congregate. “We define a foundry as a collection of annotation layers that have something in common: they may have been simply produced by the same tool, or at least they elaborate on the same theoretical view of the data (in the case of foundries containing hierarchical annotation layers building upon one another)” (Bański et al., 2011). KorAP provides two tokenization layers as part of the base foundry for every document – one with a rather aggressive splitting approach (‘greedy’) and one that only interprets white spaces as token boundaries (‘conservative’). Apart from the tokenization layers, the base foundry contains two layers that store sentence and paragraph boundaries.

```
<raw_text docid="WPD_AAA.00001">
  <metadata file="metadata.xml"/>
  <text>A bzw. a ist [...]</text>
</raw_text>
```

Figure 1: Extract from a file storing primary text.

```
<metadata docid="WPD_AAA.00001">
  <doc file="text.xml" />
  <foundry name="base"
        path="base/" />
</metadata>
```

Figure 2: Extract from a document metadata file.

The first deviation from the standard Lucene indexing process is that KorAP does not want Lucene to perform the tokenization during indexing because it uses tokenizations produced externally and independently of the indexing process. One way to achieve this would be to re-implement the tokenization algorithm(s) embedded into a Lucene analyzer so that the Lucene tokenizer exactly re-produces the results of the external tokenizer. This would comply with the Lucene-native solution, but not allow the engine to work with a tokenization from which only the results are known, without the algorithm that produced it, e.g. from a closed-source tool or uploaded by a user.

The KorAP implementation uses a Lucene analyzer that takes as input the location of an XML file instead of the actual primary text. That file lists the spans or tokens (cf. Figure 3) and provides a pointer to the primary text; this is implemented indirectly through the foundry metadata. The KorAP analyzer thus parses three files – the tokenization layer, the foundry metadata, and the primary text file – and generates the tokens for indexing. This method yields a Lucene index based on the tokens defined in the tokenization layer file without introducing any own tokenization logic during the indexing.

```
<layer docid="WPD_AAA.00001">
  <spanList>
    <span from="64" to="67" />
    <span from="68" to="73" />
  </spanList>
</layer>
```

Figure 3: Extract from an annotation file segmenting the primary text.

4.2 Annotations

On an abstract level, the indexing engine interprets all annotations in the same way: as character spans to which values are assigned. From an implementation point of view, an annotation either only defines a span by character offsets or it additionally provides a value to that span, e.g. a part-of-speech tag. In the former case, the actual character sequence is taken as the term to be indexed, while in the latter case, the tag value is the relevant information.

In the KorAP XML representation, a span has an optional feature structure in an `<fs>`-element. Figure 4 shows an annotation where different values (lemma, certainty, and morpho-syntactical tag) are assigned to a span. If a `<span>`-element contains no `<fs>`-element, it is a purely segmenting annotation (cf. Figure 3).

4.3 Concurrent Tokenizations

KorAP shifts the definition of a token away from the engine and towards the tools and users that provide tokenizations. It is designed to accept any tokenization logic, leaving the judgement about its meaning and usefulness entirely to the user.

```
<span from="0" to="1">
 <fs type="lex">
  <f name="lex">
   <fs>
 <f name="lemma">A</f>
 <f name="certainty">0.780715</f>
 <f name="ctag">NN</f>
   </fs>
  </f>
 </fs>
</span>
```

Figure 4: A span annotated with a feature structure.

This flexibility resolves an issue that has appeared whenever text analysis is based on tokens: different results in tokenization are the norm rather than the exception (Chiarcos et al., 2009). While providing two basic tokenizations, KorAP allows for custom tokenizations as well, so that users can opt to base their queries either on one of the built-ins or on any other tokenization for their research. The user always specifies a tokenization layer to query, although the user interface can generally set a default when she has not. Also, multiple tokenizations can be queried at the same time in one search request.

4.4 Architecture and Implementation Details

4.4.1 Storing Higher-Level Information

As described before, an inverted index is always bound to some definition of tokens. In Lucene, a token object has an optional *Payload* field that holds an array of bytes. This is the place where higher-level information can be stored, for instance the sentence in which a specific token instance has occurred; cf. Brin and Page (1998). This is implemented by assigning IDs to such above-token spans: when parsing the XML file that holds sentence boundary information, the analyzer derives an ID for each sentence that encodes a reference to the annotation layer, a pointer to the primary text, and positional information.

In order to find, for instance, two or more tokens that occur in the same sentence, the search engine matches their sentence IDs, stored in the payloads. Because the IDs represent the positional order between sentences, the engine can as well search for a word occurring in two subsequent sentences. This allows for more complex queries, for instance to find two words that occur in the same verb phrase, but in different noun phrases: the IDs for the verb phrase have to match and the IDs for the noun phrase have to differ.

Inverted indexes are efficient only when querying on the base of token strings. However, annotations have to be searchable efficiently, too, which is why they cannot just be stored as payloads on tokens. This is where the previously mentioned abstraction of segmenting and labelling annotations comes into play: by applying the same techniques as presented for tokens, further indexes can be built using keys that are not (only) based on the actual text character sequences, but rather on the vocabulary applied by the annotation tool or human annotator. For example, an index can be introduced that uses part-of-speech tags as keys and lists the occurrences for each tag. This method can also be applied for constituencies, lemmas etc.

Different indexes can be queried separately and in parallel. Using the MapReduce paradigm (Dean and Ghemawat, 2004), queries are combined with queries on other indexes and merged using the appropriate set operation (see Section 4.4.2). The efficiency and scalability of this approach depends mainly on the vocabulary, i.e. the number of index keys, used in a given annotation, and will be evaluated during further development.

4.4.2 Map and Reduce in Practice

KorAP aims to be scalable, which means in practice that increasing corpus sizes must not lead to a growth in computational cost to an extent that could not compensated by increasing hardware resources. In order to achieve this goal, KorAP parallelizes as many operations as possible, applying the MapReduce paradigm: each query is divided into independent sub-queries that can be processed independently of each other and eventually, the results of these sub-queries are merged to the final result. This allows for parallelization and distribution of work load to many processors, distributed across different machines, making the platform scale up for very large corpora by adding more machines to the system.

The Lucene index structure is well suited for this strategy because any index can be split into smaller segments that can be distributed across different disks and machines. A central header node divides incoming queries into atomic sub-queries and distributes them to the machines in the cluster that hold the actual indexes, the worker nodes. Each worker node queries its index(es) and sends back a set of results to the header node where the result sub-sets are merged to the full result set.

For instance, if the index of a full corpus is to be distributed across five machines, five indexes are created, each based on a different (possibly overlapping) sub-set of the total document collection. When a simple term query comes in to the head node, e.g. "find all occurrences of word 'A'", the head node forwards that request to all five worker nodes. Each of them searches its respective sub-index and returns a set of occurrences to the head node. The head node joins the five result sets in order to produce the final result.

For a more complex sample query like "find word 'A' in sentences that also contain word 'B'", the query is split into two queries "find word 'A'" and "find word 'B'" and these two queries are sent to the worker nodes; this is the map step in MapReduce terminology. After the worker nodes have returned their results for both queries, each of the sub-results for 'A' and 'B' are joined to two sets. These sets are intersected – corresponding to the Boolean `AND`-operator – so that only results from the two sub-sets end up in the final result set that share the same sentence ID; the reduce step in MapReduce speak. Other Boolean operators – `OR` and `NOT` – can be realised through the union and difference set operations. Some queries could limit the search to a sub-set of the full corpus (virtual corpora, cf. Kupietz et al. (2010)) so that only those partitions have to be addressed that hold the relevant sub-corpora.

In the KorAP implementation, each query is at first divided into the annotation layers that are queried. In a second step, each of the single terms is isolated. The latter forms the smallest unit of a query and corresponds to a callable thread at runtime. The reducing is performed in the inverse order: the term results for a layer, returned by the different threads, are joined according to the conjunctions specified in the query (`AND`, `OR`, `NOT`). Subsequently, the layer results are joined to form the total result set.

In order to speed up actual query performance upon intensive querying, index partitions can be stored redundantly, so that similar queries can be handled by different machines in parallel. This allows scaling up the platform to an almost unrestricted extent: when requests that query a certain partial index turn out not to be answered fast enough, another machine can be added to the cluster that helps out at frequent tasks.

Another potential bottleneck is the head node, although its main tasks – atomizing and distributing queries – are computationally less costly. However, multiple head nodes can parallelize these tasks as well when necessary.

5 Results

Benchmarking indexing and querying performance of a platform like KorAP has to consider a large number of factors, some of which are mutually dependent. Corpora of relevant size do not fit on a single hard disk, so that the file system choice matters as well as the distribution strategy, across multiple hard disks and across a network. Also, RAM capacity could lead to unexpected result because the system might not show significant speed reductions with increasing data amounts as long as it can hold the indexes in memory, but as soon as the platform has to store intermediate results on a hard disk, processing performance could decrease all of a sudden.

Performance comparisons between solid-state disks (SSDs) and magnetic hard disks (HDDs) could show surprising impacts as well because the physical location and ordering of data on the disk plays an important role. If data is dispersed across the disk, an HDD reading head has to jump across the platter while SSDs do not suffer speed losses in that scenario. When reading a sequence of bytes, the reading speed might not differ significantly though.

In order to find comparable statistics that yield insights about how the presented Lucene-based implementation scales in relation to the data volume, indexes from corpora of different sizes were created, all of them sub-samples of DeReKo (IDS, 2011). The number of documents in each test corpus are reported in Table 1.

#	#Documents	Index Size
1	29,704	1.6 GByte
2	196,854	16 GByte
3	512,542	33 GByte
4	974,722	42 GByte
5	1,487,264	75 GByte
6	2,080,111	76 GByte
7	3,597,079	151 GByte

Table 1: Sample corpora – sub-sets of the DeReKo corpus – were used to test query performance (cf. Tables 2, 3, 4, 5). The reported sizes sum over three tokenization layers and one annotation layer contained in the indexes.

We have applied two different tokenization algorithms, a sentence splitter, and TreeTagger (Schmid, 1994), including its own tokenizer, on the full corpus. Our self-made tokenizers follow different approaches in disputable cases such as hyphenations within words; the 'conservative' method treats such items as one token, while 'greedy' splits instances like "Ski-WM" into three tokens. By default, 'conservative' has been used in search.

The following queries were applied to all the sample corpora:

1. Simple token search: *'Alphabet'.*
2. Concurrent tokenization: *'Ski-WM' in conservative tokenization and 'Ski' in greedy.*
3. Inter-layer search: *'Buchstabe' and 'Alphabet' occurring in one sentence.*
4. Wildcard search: *All tokens that start with 'Alpha' ('Alpha*').*
5. Part-of-speech (POS) search: *'Alphabet' tagged as noun (NN) (both tokenized and tagged by TreeTagger).*
6. Inter-layer annotation search: *Token 'Alphabet' (conservative), tagged as Noun (NN) by TreeTagger.*
7. High frequency: *Tokens tagged as personal pronoun (PPER) that start with 'er'.*

Tables 2, 3, 4, and 5 report the average response times for the different queries executed three times on each of the sample corpora listed in Table 1, every time disregarding the first call in order to allow caching to take effect. For each match, the surrounding sentence was returned as context; for two matches within one context sentence, that sentence was counted as one hit only. Therefore, the number of hits does not exactly represent the number of matches; this is especially important because searches were aborted after 1 million matches where necessary, yielding different numbers of hits.

	Query 1		**Query 2**	
Sample	reponse	#hits	reponse	#hits
1	0.038s	10	0.021s	0
2	0.277s	996	0.198s	53
3	0.301s	1,078	0.291s	136
4	0.135s	82	0.578s	472
5	0.226s	1,160	0.616s	608
6	0.225s	380	0.418s	262
7	0.400s	1,550	0.803s	870

Table 2: Queries 1 and 2 for all sample corpora (Table 1), reporting response time and number of hits.

	Query 3		**Query 4**	
Sample	reponse	#hits	reponse	#hits
1	0.034s	0	0.397s	837
2	0.057s	89	0.201s	2,575
3	0.131s	89	0.351s	3,308
4	0.031s	1	0.175s	1,167
5	0.102s	90	0.487s	4,475
6	0.055s	8	0.287s	2,214
7	0.140s	98	0.828s	7,526

Table 3: Queries 3 and 4 for all sample corpora (Table 1), reporting response time and number of hits.

The indexes were stored on a Linux machine with 48 CPU cores, 256 gigabytes of memory, on an Ext4 file system in a storage area network (SAN) on a RAID-5-volume. In order to retrieve the context properly, the Lucene `SpanQuery` class has been used in all cases. The present indexing implementation has been based on Lucene version 3.6.0.

Comparable index building times are not available here because they depend on factors like the XML parser, file system performance, and network load that lie beyond the scope of this pa-

	Query 5		Query 6	
Sample	reponse	#hits	reponse	#hits
1	0.015s	10	0.043s	9
2	0.066s	833	0.110s	564
3	0.044s	915	0.158s	624
4	0.018s	70	0.015s	52
5	0.055s	985	0.087s	676
6	0.026s	374	0.039s	279
7	0.068s	1,369	0.098s	964

Table 4: Queries 5 and 6 for all sample corpora (Table 1), reporting response time and number of hits.

	Query 7	
Sample	reponse	#hits
1	0.333s	8,489
2	4.234s	186,969
3	9.032s	421,229
4	10.721s	522,118
5	20.362s	910,038
6	19.543s	892,494
7	19.544s	894.905

Table 5: Query 7 for all sample corpora (Table 1), reporting response time and number of hits; search was aborted after 1 million matches.

per. However, parsing the XML input files and building indexes for the corpora reported in Table 1 took between 4 and 60 hours.

Anyway, indexing is not the most time-critical part in the KorAP scenario: new DeReKo versions are released only twice a year. At those points, they can be indexed in background while users can still use the previous release. The search, on the other hand, is expected to deliver instant results whenever possible. There might be compromises necessary in case of very complex queries, but the querying side is where KorAP places emphasis on performance. In short: the engine has not been optimized to build indexes fast, but to be queried fast.

The reported response times reveal one result very cleary: the querying time depends on the number of hits more than on the size of the corpora or indexes. In a simple token query (Query 1), sample corpus 3 contains many more hits than sample 4 which results in a threefold response time, despite having only approximately half of the size. Less obvious differences between the corpora such as document sizes, vocabulary and other factors probably play a role too, but the number of hits seems to be most significant.

Not surprisingly, more complex queries take longer to process, but reply times still increase less than linearly in relation to corpus size. Intersections only increase response significantly when there are very many hits either, as in Query 7 (Table 5). In order to avoid intersections with large result sub-sets, one approach is a smarter distribution of indexes, so that joins can be performed at an early stage with smaller sets, before the final reduce step is performed on the full set. Another factor is query optimization so that complex queries are rewritten in order to avoid intersections and set differences whenever possible.

6 Summary & Outlook

We have used the Lucene engine to develop an indexing module for the KorAP corpus analysis platform. We have implemented a Lucene analyzer class that handles pre-analysed texts and corpora with multi-level stand-off annotations. The platform shifts away the task of segmenting, tokenizing, and analysing corpora towards the users and external tools and is ready to include new annotations of different kinds. We have presented a way to apply inverted indexes within a MapReduce-like environment, parallelizing tasks and making the platform scalable and ready to process very large corpora. Implicitly, this work demonstrates that techniques and software from the related field of IR can successfully be applied for linguistic search tasks.

Acknowledgements

This work has been made possible by the *Institut für Deutsche Sprache* (IDS) in the context of the KorAP project, funded by the *Leibniz-Gemeinschaft* and is a direct result of the work with the other KorAP core team members Piotr Bański and Elena Frick. Further IDS colleagues from within and outside the KorAP team have contributed important theoretical advice and practical help: Cyril Belica, Peter Fankhauser, Marc Kupietz, Roman Schneider, Oliver Schonefeld, and the Cosmas-II-Team: Franck Bodmer, Peter Harders, Helge Krause.

References

P. Bański, C. Belica, H. Krause, M. Kupietz, C. Schnober, O. Schonefeld, and A. Witt. 2011. KorAP data model: first approximation, December.

P. Bański, Peter M. Fischer, E. Frick, E. Ketzan, M. Kupietz, C. Schnober, O. Schonefeld, and A. Witt. 2012. The new IDS corpus analysis platform: Challenges and prospects. In *Proceedings of LREC-2012*, Istanbul, May.

F. Bodmer. 2005. COSMAS II. Recherchieren in den Korpora des IDS. *Sprachreport*, 21(3):2–5.

S. Brin and L. Page. 1998. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1-7):107–117.

C. Chiarcos, J. Ritz, and M. Stede. 2009. By all these lovely tokens...: merging conflicting tokenizations. In *Proceedings of the Third Linguistic Annotation Workshop*, pages 35–43. Association for Computational Linguistics.

J. Dean and S. Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, December.

R. Fiehler and P. Wagener. 2005. Die Datenbank Gesprochenes Deutsch (DGD)-Sammlung, Dokumentation, Archivierung und Untersuchung gesprochener Sprache als Aufgaben der Sprachwissenschaft. *Gesprächsforschung-Online-Zeitschrift zur verbalen Interaktion*, 6:136–147.

S. Ghodke and S. Bird. 2008. Querying linguistic annotations. In *Proceedings of the 13th Australasian Document Computing Symposium*, pages 69–72.

S. Ghodke and S. Bird. 2010. Fast query for large treebanks. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 267–275. Association for Computational Linguistics.

IDS. 2011. Deutsches Referenzkorpus / Archiv der Korpora geschriebener Gegenwartssprache 2011-II.

D. Janus and A. Przepiórkowski. 2007. Poliqarp: An open source corpus indexer and search engine with syntactic extensions. In *Proceedings of the 45th Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions*, pages 85–88. Association for Computational Linguistics.

A. Kilgarriff. 2007. Googleology is bad science. *Computational Linguistics*, 33(1):147151.

D.E. Knuth. 1997. *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, 3rd edition.

M. Kupietz, C. Belica, H. Keibel, and A. Witt. 2010. The German reference corpus DeReKo: a primordial sample for linguistic research. In *LREC 2010 Main Conference Proceedings. Malta*.

Lucene, 2012. *Apache Lucene – Index File Formats*. The Apache Software Foundation, April.

M. McCandless, E. Hatcher, and O. Gospodnetić. 2010. *Lucene in Action*. Manning Publications Co., 2nd edition, July.

R. Perkuhn, H. Keibel, and M. Kupietz. 2012. *Korpuslinguistik*. Fink. (UTB 3433), Paderborn.

H. Schmid. 1994. Probabilistic part-of-speech tagging using decision trees. In *Proceedings of international conference on new methods in language processing*, volume 12, pages 44–49. Manchester, UK.

R. Schneider. 2012. Evaluating DBMS-based access strategies to very large multi-layer corpora. In *Proceedings of the LREC 2012 Workshop: Challenges in the management of large corpora*. European Language Resources Association (ELRA).

J. Sinclair, 2004. *Trust the text*, chapter 1. Routledge, Milton Park, UK.

A. Sokirko, 2003. *A technical overview of DWDS/Dialing Concordance*.

Henry S. Thompson and David McKelvie. 1997. Hyperlink semantics for standoff markup of read-only documents. In *Proceedings of SGML Europe*.

M. Volk. 2002. Using the web as corpus for linguistic research. *Tähendusepüüja. Catcher of the Meaning. A Festschrift for Professor Haldur Oim*.

A. Zeldes, J. Ritz, A. Lüdeling, and C. Chiarcos. 2009. Annis: A search tool for multi-layer annotated corpora. In *Proceedings of Corpus Linguistics*, pages 20–23.

C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. 2001. On supporting containment queries in relational database management systems. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, volume 30, pages 425–436. ACM.