Nils Diewald, Eliza Margaretha

Krill: KorAP search and analysis engine

1 Introduction

KorAP[1] (*Korpusanalyseplattform*) is a corpus search and analysis platform for handling very large corpora with multiple annotation layers, multiple query languages, and complex licensing models (Bański et al., 2013a). It is intended to succeed the COSMAS II system (Bodmer, 1996) in providing DeReKo, the German reference corpus (Kupietz and Lüngen, 2014), hosted by the Institute for the German Language (IDS).[2] The corpus consists of a wide range of texts such as fiction, newspaper articles and scripted speech, annotated on multiple linguistic levels, for instance part-of-speech and syntactic dependency structures. It was reported to contain approximately 30 billion words in September 2016 and still grows continually.

Krill[3] (*Corpus-data Retrieval Index using Lucene for Look-ups*) is a corpus search engine that serves as a search component in KorAP. It is based on Apache Lucene,[4] a popular and well-established information retrieval engine. Lucene's lightweight memory requirements and scalable indexing are suitable for handling large corpora whose size increases rapidly. It supports full-text search for many query types including phrase and wildcard queries, and allows custom implementations to cope with complex linguistic queries.

In this paper, we describe Krill and how its index is designed to handle full-text and complex annotation search combining different annotation layers and sources of very large corpora. The paper is structured as follows. Section 2 describes how a search works in KorAP (starting from receiving a search request until returning the search results). Section 3 explains how corpus data are represented and indexed in Krill. Section 4 describes various kinds of queries handled by Krill and how they are processed for the actual search on the index. The Krill response format containing search results is described in Section 5. We present related and further work in Section 6 and 7 respectively. The paper ends with a summary.

2 Search Flow

The KorAP architecture's design is based on a microservice architecture comprising small independent components that are easy to extend, to replace and to maintain (Diewald et al., 2016). Figure 1 illustrates the KorAP architecture and the interactions

[1] https://korap.ids-mannheim.de/
[2] http://www.ids-mannheim.de/
[3] https://github.com/KorAP/Krill
[4] https://lucene.apache.org/

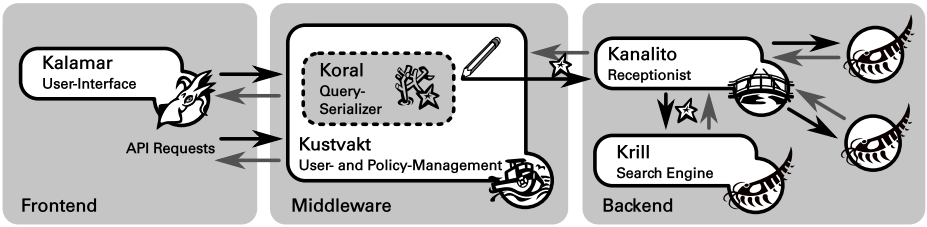

Figure 1: The KorAP Architecture

among its components. In addition to the search engine Krill, the components include a user interface (Kalamar), a query serializer (Koral), a user and resource policy management system (Kustvakt) and a *receptionist service* (cf. Büttcher et al., 2010, ch. 14.1.1) for the search engine to provide parallel search (Kanalito).

A KorAP search process starts by sending a query in a particular query language, either using the Kalamar frontend or a direct API request to Kustvakt. Supported query languages include COSMAS II (Bodmer, 1996), ANNIS (Rosenfeld, 2010), and Poliqarp (a CQP variant; Przepiórkowski et al., 2004).

The query is then serialized by Koral resulting in a generic representation as KoralQuery (Bingel and Diewald, 2015). Kustvakt may relay the KoralQuery to a single Krill instance to conduct a search or to Kanalito for a parallel search of distributed Krill instances. The actual search is performed in Krill, and query results are eventually returned to the API endpoint and may be displayed in Kalamar.

Kustvakt is the API provider of KorAP managing the interaction of all components. One of its primary tasks is to monitor user access to resources, and thus to guarantee the retrieval of resources with respect to their intellectual property rights before commencing a search (Bański et al., 2014). When a user request involves unauthorized resources, Kustvakt may rewrite the corresponding KoralQuery in order to limit the query to only those resources available to that user. Besides, it may inject values from user preferences such as default annotation sources for each annotation layer. Because the central user and license mechanism is done in Kustvakt and not in Krill, redundancy in distributed search and re-implementation in different search component systems can be avoided.

3 Index Representation

Krill, as it is based on Lucene, uses an inverted index[5] to provide full-text search capabilities in a corpus. In contrast to tools like *grep*, this approach does not treat textual data as a sequence of characters, but as a sequence of tokens (most often "words"). Thus, searching for these tokens provides direct access to the occurrence of these tokens in a collection of documents. An inverted index consists of a term dictionary (based on the tokens) with direct access to the associated information on the

occurrence of a token in the corpus, so-called postings lists (see Fig. 4). While most of the time the dictionary is kept in memory for fast access, postings lists are only loaded (partially) in memory, everytime a term is retrieved as part of the query process.

As a search component of KorAP, Krill indexes and provides search on primary data (i.e. corpus text), various layers of annotation data (e.g. lemma, part-of-speech, constituent annotations) and metadata of the resources. The KorAP data model (see Bański et al., 2013b) separates primary data and annotations (i.e. a *standoff* architecture) to allow for multiple, potentially concurring layers of annotations (see Fig. 2).[6] This model is the foundation of the KorAP input format *KorapXML* as well as the foundation of the Krill index design.

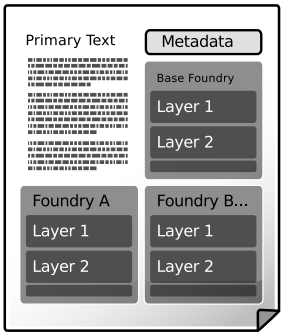

Figure 2: KorAP data model

Metadata provides information on the *document-level* (cf. Zobel and Moffat, 2006), such as author information of a text or the publication date. The data structure of the term dictionary for metadata depends on the stored data type and the expected matching operations. For example, publication dates may need to be stored as numerical data to provide support for range queries (e.g. "Search in all documents published since 1786"), while titles may be stored as full-text searchable data (e.g. "Search in all documents containing the word 'Reise' in the title"), and organizing the term dictionary for string data using hashes may be efficient, but may also prevent access using regular expressions.

Metadata fields in Krill can have the following types:[7]

date	Date field (e.g. publication date)
int	Numerical integer field (e.g. number of tokens)
keyword	Multiple strings (e.g. keyword tags)
store	Retrieve-only value (e.g. associated file reference)
string	Fixed string (e.g. text identifier)
text	Tokenized full-text field (e.g. text title)

Annotation data provides information on the *word-level* of the text and is subdivided into *foundries* denoting the resource of a certain annotation (e.g. the annotation tool used to generate the annotation) that may contain multiple annotation *layers*. In KorAP, a user can search for an annotation in a specific foundry and layer, by adding a prefix to the query term, specifying the requested annotation, for example `[mate/l=sun]` to

[5] For a brief introduction to inverted indices for full-text search, see Zobel and Moffat (2006).

[6] Although Krill supports overlapping annotations, at the moment it is limited to one stream of tokens (i.e. a single tokenization for all layers).

[7] Currently, metadata fields are defined by DEREKO. In the future, Krill will support arbitrary metadata fields of the given types.

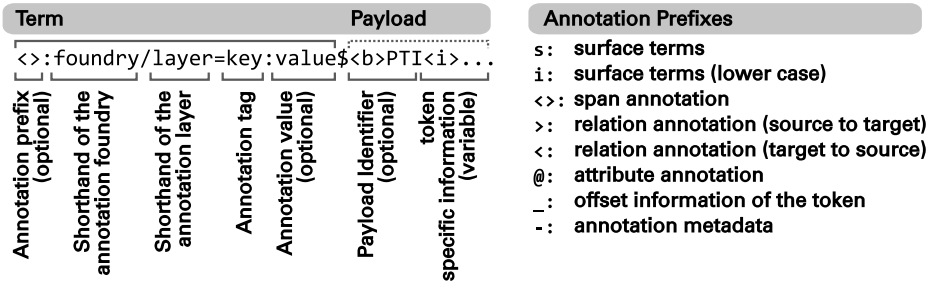

Figure 3: Term structures for annotations in the Krill index

search for the term "sun" in the lemma layer of the *Mate*-foundry using the Poliqarp+ query notation (KorAP's extension to Poliqarp).

To distinguish between annotations in the index, all annotation terms in the term dictionary have foundry and layer prefixes (see Fig. 3, left). Annotations in Krill can have the following types, specified as optional additional type prefixes to the dictionary terms (see Fig. 3, right):

Token Token-level information terms
(e.g. the surface form of a term or the lemma)

Span Span-level information terms
(e.g. sentences or nominal phrases)

Relation Relational information spanning between terms and/or spans
(e.g. dependency relations)

Attribute Annotations to tokens, spans, or relations
(e.g. the `href`-attribute of an HTML anchor element)

Further annotation types are being considered, for example to describe punctuations.[8]

The document identifier and the starting position of the annotation are stored in the postings list, along with additional retrievable information encoded in byte streams (so-called *payloads*),[9] for example the length of a span or the annotation a relation refers to.

Additional information supported by all annotation types includes a leading byte for term type identification, a unique token identifier, and a byte value indicating the level of confidence of the annotation source.

Character offsets of tokens (relevant for matching highlights, see Sec. 5) are stored once per token position. Span terms may store additional character offset information in payloads, to include surrounding non-token characters, for example punctuation and quotation marks.

[8] Krill does not support punctuation search yet. In the future, punctuations will be indexed as attachements to surrounding tokens in order to keep searching for sequences of words simple.

[9] The documentation of payload byte streams is available at `https://github.com/KorAP/Krill/blob/master/misc/payloads.md`.

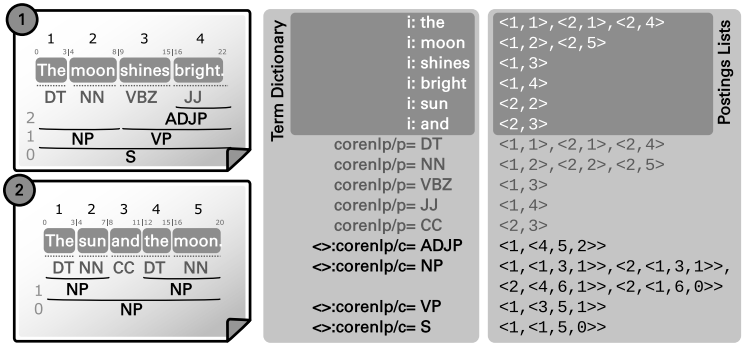

Figure 4: Example index representation of two documents, showing the term dictionary and the postings lists

Figure 4 exemplifies the structure of the Krill index for two documents. Both documents have a sequence of tokens (*token stream*), a token-level annotation and a span-level annotation. The term dictionary contains all annotations including prefixes. The associated postings lists are illustrated as lists of tuples starting with the identifier of the text in which the annotation occurs, followed by occurrence-specific information, that vary according to the annotation type. The surface terms (e.g. `i:moon`) and the token-level annotations (e.g. the part-of-speech annotation `corenlp/p=DT`) store their token position in the token stream. The span-level annotations (e.g. the constituency annotation `<>:corenlp/c=S`) store their start and end positions, and the level in the tree hierarchy, partially encoded in payloads.

An annotation as in Figure 4 would be encoded in the following structure:

```
<>:corenlp/c=NP$<b>64<i>0<i>8<i>2<b>1
```

This structure is expected by Krill when processing token streams. The prefix `<>` denotes the span type, `corenlp/c` denotes the foundry and the layer, and `NP` denotes the annotation value. `$` splits the term dictionary part and the postings list part whose structure may differ depending on the information needed to be stored. Everything that follows `$` is stored per token as a byte stream. The information in the angular brackets defines the data type to decode the following information (`<b>` for byte, `<i>` for integer etc.). The first byte (`<b>64`) introduces the payload identifier for denoting a span-level annotation, the following two integers encode the character offsets of the annotation (`<i>0` to `<i>8`), the next integer contains the token position after the end of the span (`<i>2`), and the final byte represents the depth of the annotation in a hierarchy (`<b>1` means the annotation is a direct child to the root).

In KorAP, a separate internal preprocessing pipeline is used to enrich corpus data (based on the I5 format of DeReKo; see Lüngen and Sperberg-McQueen, 2012) by adding standoff annotations from multiple foundries (including CoreNLP, Mate or Xerox

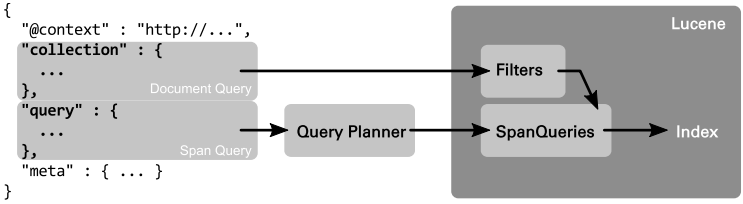

Figure 5: KoralQuery translation into Lucene applicable queries

parsers) resulting in KorapXML (Bański et al., 2012). A `base` foundry provides the minimum annotation necessary for Krill, including sentences, paragraphs, and text boundaries. The KorapXML data can be transformed into a JSON document consisting of a single token stream of the Krill term structure described above by using a conversion tool.[10] The resulting JSON document can be indexed by Krill.

A Krill index is not static and is updated every time a new document is added or deleted. Deletion is done by maintaining a list of deleted documents that are excluded from every virtual corpus requested (see Sec. 4.1). The documents will be purged from the index regularly as a defragmentation of the index. Moreover, Krill uses unique text identifiers in the metadata for updates: documents are updated by deleting the old version of a document and adding a new one to the index. As a consequence, every modification of the document, such as an addition of annotations, will result in a new version of the whole document.

4 Query Processing

KorAP is designed to handle various kinds of queries from existing corpus query languages. These queries of different syntaxes are represented in a common format by the KoralQuery protocol (see Sec. 2 and Bingel and Diewald, 2015) serialized in JSON-LD (Sporny et al., 2014). KoralQuery describes complex linguistic queries as nested objects with various types and operations. Krill is the reference implementation for KoralQuery consumption, aiming for completely supporting it.

On the root level, KoralQuery distinguishes between *document queries* and *span queries*. Document queries allow to specify a virtual corpus (i.e. a defined subcollection of documents) by means of document-level metadata constraints (see Bański et al., 2013b), while span queries define a textual span to search in the virtual corpus on the word-level.

Figure 6 shows a KoralQuery document. The `query` section is a serialization of the query `[orth=sun][][orth=moon]?` (Poliqarp notation). The `collection` section defines a virtual corpus to limit the search to all documents where the `author` metadata field contains the term “Goethe”, the `pubDate` field contains a date since 1786, and

[10] `https://github.com/KorAP/Korap-XML-Krill`

```
{
  "@context" : "http://korap.ids-mannheim.de/ns/koral/0.3/context.jsonld",
  "collection" : {
    "@type": "koral:docGroup",
    "operation": "operation:and",
    "operands": [{
      "@type": "koral:doc",
      "key": "author",
      "match": "match:contains",
      "value": "Goethe"
    },{
      "@type": "koral:docGroup",
      "operation": "operation:and"
      "operands": [{
        "@type": "koral:doc",
        "key": "pubDate",
        "match": "match:geq",
        "type": "type:date",
        "value": "1786"
      },{
        "@type": "koral:docGroup",
        "operation": "operation:or"
        "operands": [{
          "@type": "koral:doc",
          "key": "title",
          "match": "match:contains",
          "value": "Reise"
        },{
          "@type": "koral:doc",
          "key": "title",
          "match": "match:contains",
          "value": "Wahlverwandtschaften"
        }],
      }]
    }]
  },
  "query": {
    "@type": "koral:group",
    "inOrder": true,
    "operation": "operation:sequence",
    "operands": [{
      "@type": "koral:token",
      "wrap": {
        "@type": "koral:term",
        "key": "sun",
        "layer": "orth",
        "match": "match:eq"
      }
    },{
      "@type": "koral:token"
    },{
      "@type": "koral:group",
      "operation": "operation:repetition",
      "boundary": {
        "@type": "koral:boundary",
        "max": 1,
        "min": 0
      },
      "operands": [{
        "@type": "koral:token",
        "wrap": {
          "@type": "koral:term",
          "key": "moon",
          "layer": "orth",
          "match": "match:eq"
        }
      }]
    }]
  }
}
```

Figure 6: A KoralQuery document

Figure 7: Bit vector calculation of the virtual corpus as defined in Figure 6 based on five documents

a `title` field that contains either the term "Reise" or "Wahlverwandschaften" (see Diewald and Bingel, 2015, for the KoralQuery specification).

To process a query, Krill parses and validates the KoralQuery and transforms the virtual corpus into applicable Lucene Filters and span queries into applicable Lucene SpanQueries (see Fig. 5). The span query may need to be rewritten into an intermediate query tree according to a query plan to be applicable. The following sections describe these steps.

4.1 Document Queries

In Krill, every search is limited to a virtual corpus, therefore the virtual corpus is prepared first based on the nested constraints in KoralQuery. Each constraint is described by a key (e.g. "author" for the name of the author of a document), potentially a value (e.g. "Goethe" as the author's name), a matching operator, expressing how the requested value has to match with the document's value (e.g. as a prefix or a postfix) and the type of the constraint (e.g. if the constraint represents a string match, a date range, a regular expression; cf. Figure 6, line 7–10). These constraints can be nested in groups of boolean operations. If no constraint is defined, the virtual corpus contains all available documents.

For each metadata term in the term dictionary, a postings list of the documents is stored in which the term occurs (see Sec. 3). As metadata information is stored on the document level, postings lists for metadata information can simply be treated as bit vectors with one bit per document in the corpus. In case a metadata term exists for one document, the associated bit is set. Using bitwise boolean operations, arbitrary complex virtual corpora can be constructed. Figure 7 shows the calculation of the virtual corpus for the nested document query in Figure 6 based on five documents, using bitwise *or* (`|`) and bitwise *and* (`&`). The resulting virtual corpus can be represented as a bit vector itself, allowing for space-efficient caching.

The actual implementation of document level postings lists varies depending on several factors (e.g. the density of documents or if a postings list is stored on disk or cached in memory). For sparse cached postings lists (e.g. only a fraction of the corpus' documents contain the word "Wahlverwandtschaften" in the title field) Lucene, starting with version 5, implements bit sets as so-called *Roaring bitmaps* focusing on efficient compression and fast bit operations (Chambi et al., 2016). Range queries as mentioned

in the example above are precompiled as a disjunction of the postings lists of all terms in the given range optimized using indexed postings lists of sub ranges (see Schindler and Diepenbroek, 2008, pp. 1957f).

Operations on the virtual corpus include counting the number of texts in the virtual corpus (equivalent to the cardinality of the bit vector), aggregating stored numerical data (e.g. number of tokens, sentences, paragraphs in the virtual corpus), and grouping of statistical data by certain metadata fields (e.g. the distribution of sentences per genre in the virtual corpus). Document queries restrict span queries to documents that are elements of the virtual corpus (see next section).

4.2 Span Queries

In addition to the *document-level* term index for metadata information, the *word-level* term index for tokenized textual data requires more information than the existence in a document, for example, the position of a certain token in a text (cf. Sec. 3). To find, for example, a sequence of two words like "the sun" in a corpus, both terms are searched in the term dictionary and their associated postings lists are analyzed in parallel, to find matching documents and consecutive token positions that indicate a sequence of these two tokens.

Krill supports various query constructs as specified in KoralQuery. Analogously to document queries, these query constructs can be nested and become arbitrarily complex. The resulting query tree consists of leaf nodes with term look-ups, returning the span information of a term, and inner nodes of operations that can be applied on the nested spans.

The result of a span query is always a span, meaning that it always contains information on the start position, the end position, and additional optional information in collected payloads. Operations may use this information to compare nested spans, for example, if two tokens are consecutive in an operation requiring a sequence of two tokens. Payloads may also be passed further to nested operations. Moreover, operations are capable of adding new information to the span's payloads.

If some spans satisfy the requirements of a span query, new spans are created and returned, for example in the case of the consecutive tokens, a span starting at the first token and ending at the last one is created as the resulting span.

Lucene implements such operations as so-called *SpanQueries*. While document queries may use fast bit operations for matching, SpanQueries always iterate sequentially over the postings lists of every term that is part of the query as well as every temporary postings list, that is the result of a span operation.[11]

[11] Lucene uses deterministic *SkipLists* (Pugh, 1990; Munro et al., 1992) to improve performance of moving forward to specific documents in postings lists, for example to skip documents that are not part of the virtual corpus.

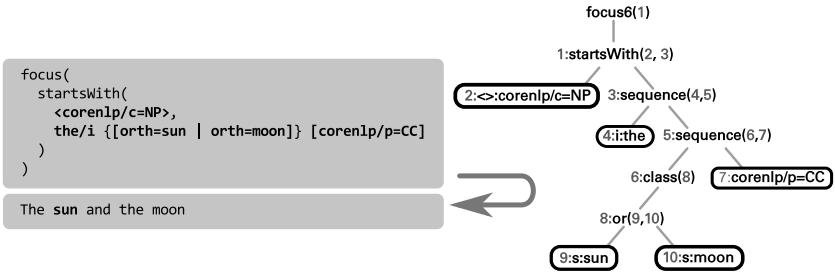

Figure 8: A nested span query with its query tree visualization

To be able to fully support KoralQuery, the set of Lucene SpanQueries was largely extended and new concepts (like *classes*) were introduced.[12] Supported query constructs in Krill can be categorized as follows:

Term queries index look-up of tokens, spans, relations, and attributes (see p. 4), including regular expressions

Comparison queries include logical operations (*and*, *or*, *not*), distance (matching two spans with a defined number of tokens or spans in between), position (matching one span in a positional relation to another, e.g. embedding or overlapping), repetition (matching one span with a defined sequential repetition), etc.

Span modification queries include extensions to left or right, etc.

Class queries include class setting and focus

Figure 8 shows a query in Poliqarp+ notation, searching for a sequence of the word "the" (case-insensitive), followed either by the word "sun" or "moon", and ending with a coordinating conjunction (`CC`) annotated by the `corenlp` foundry. As a wrapping constraint, the sequence needs to be at the beginning of a nominal phrase (`NP`) as annotated by the `corenlp` foundry. In addition, a fragment of the query – the *or*-relation of "sun" and "moon" – is marked as a class (using curly brackets) and at the root of the query tree, this class is focused, meaning a match would return the span of "sun" or "moon" without their contexts.

After the query is serialized as KoralQuery and transformed into Lucene SpanQueries, the resulting query tree has the structure as illustrated on the right side of Figure 8. Term look-ups are pictured as leaf nodes in white boxes.

[12] Recent versions of Lucene support a wider variety of query constructs, that were already introduced to Krill, such as *SpanWithinQuery*.

4.3 Query Planning

Because the constructs of the KoralQuery protocol have their origins in the various corpus query languages covered by Koral that were developed for various corpus search technologies (including concepts coming from regular expressions, relational databases, XML processing and others), not all of these constructs can be directly translated into a term based search. Therefore Krill has a planning phase to rewrite queries before they get translated into actual Lucene SpanQueries.

```
[orth=sun] [] [orth=moon]?
```

The example query above[13] requests a sequence of tokens, with the first having the surface form `sun`, the second matching any token, and the last optionally matching the surface form `moon`. The *any* token of CQP-based corpus query languages matches all tokens in a token sequence (see `[]` in Evert and the OCWB Development Team, 2010, p. 11). In a linear search, this query can recognize the context of the token sequence and would accept any token as a match. An inverted index, however, has no knowledge of linear contexts of token sequences, meaning this token is not directly retrievable from the index or it would need to aggregate the postings lists of all surface terms in the term dictionary.

The query planner translates *any* tokens in sequences into either distance operations between the surrounding parts or, in case an *any* token is at the end or at the beginning of a sequence, into extension operations, that simply extend the resulting span by a certain number of tokens. For the example above, a distance of one token between `[orth=sun]` and `[orth=moon]?` is added.

```
distance(+1, [orth=sun], [orth=moon]?)
```
[14]

This query is however insufficient. Although the optional token `[orth=moon]?` at the end of the example query is retrievable in case it occurs, it needs to be reformulated to an *or*-relation operation to be processable in case it does *not* occur. For the case `[orth=moon]` occurs, `[][orth=moon]?` could be redefined as an extension of one token to the left of `[orth=moon]` and for the case it doesn't occur, as an *any* token.

```
[orth=sun] (extend(-1, [orth=moon]) | [])
```

As described above, an independent *any* token is unretrievable. Thus, the *or*-relation scope needs to be expanded to the whole query. The first case is reformulated as a distance query of one token between `[orth=sun]` and `[orth=moon]`, and the second as an extension of one token to the right of `[orth=sun]`.

```
distance(+1, [orth=sun], [orth=moon]) | extend(+1, [orth=sun])
```

[13] See the `query` section of Figure 6, line 37–69, for a KoralQuery serialization of the query.
[14] The illustration of the query plan is written as CQP based *pseudo* queries.

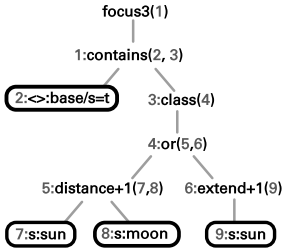

Figure 9: Rewritten span query tree for the Poliqarp query `[orth=sun][][orth=moon]?`

The right extension (`extend(+1, [orth=sun])`) may exceed the length of a matching text (imagine a text ending with the word "sun"). To prevent mismatches regarding right extensions, the query planner introduces a new constraint requiring the result being inside the text boundary as annotated by the `base` foundry. The final query tree transformed into a Lucene SpanQuery is shown in Figure 9.

Not all valid KoralQuery requests can be successfully rewritten in the way described above. In other words, not all valid queries can be answered by Krill. For example, Krill will respond with an error to a query containing just a single *any* token (`[]`), and it will respond with a warning to a query with a single optional token (e.g. `[orth=moon]?`), saying the process will ignore the optionality of the query.

The query planning phase is also used for query optimization, whenever the reorganization of the query tree can be beneficial to performance. In the example above, retrieving the postings list for `[orth=sun]` twice may result in slower performance and therefore could be rewritten to a different query plan.

5 Search Results

Although the search response format of Krill is based on JSON-LD,[15] the text snippets of each match are embedded as HTML fragments. In case of *KWIC* (Keyword in Context) results, these fragments only contain the textual content and the match marker. When retrieving further information about a match reconstructed from the inverted index, the snippet can be enriched with multiple layers of annotations inline.[16]

Figure 10 shows a match of "die Sonne" containing information about the constituency (`c`) layer of the `corenlp` foundry. API clients (like Kalamar) can easily parse this information and process it in various ways, for example to show table views for token-level information, or tree views for hierarchical spans and relations (see Fig. 11).

Classes (as introduced in the previous section) may also be used as partial highlights of a match, to map sections of the query directly to parts of the match. Figure 11 shows

[15] The search response format of KorAP is not fully specified as part of KoralQuery yet.

[16] Krill makes a distinction between token-, span- and relation-based annotations. Only concurrent retrieval of multiple token-based annotations is supported. Span- and relation-based annotations can only be retrieved separately to avoid overlapping.

```
<span class:"context-left"></span>
<mark>
  <span title:"corenlp/c:CS">
    <span title:"corenlp/c:ROOT">
      <span title:"corenlp/c:S">
        <span title:"corenlp/c:NP">die Sonne</span>
        war
        <span title:"corenlp/c:CAP">hoch und hei ß </span>
      </span>,
      <span title:"corenlp/c:S">
        ich mu ß te
        <span title:"corenlp/c:S">
          <span title:"corenlp/c:NP">meine Kleidung</span>
          erleichtern,
          <span title:"corenlp/c:S">
            die ich
            <span title:"corenlp/c:PP">
              bei der veränderlichen Atmosphäre
              <span title:"corenlp/c:NP">des Tages</span>
            </span>
            oft wechsele
          </span>
        </span>
      </span>
    </span>
  </span>
</mark>
<span class:"context-right"></span>
```

Figure 10: Match snippet with enriched annotation from `corenlp/c`

a search result in the frontend component Kalamar for the query `[corenlp/p=ART] ({1:Sonne} | {2:Mond}) []` (Poliqarp+ notation), which has two numbered classes (in curly brackets). Depending on the matching term, "Sonne" and "Mond" are underlined using different colors.

6 Related Work

Corpus search differs from typical information retrieval (e.g. web search) in terms of relevance and ranking. Whilst information retrieval aims to obtain relevant resources and ranks its results by the degree of relevance, there is typically no variation in the degree of relevance of corpus search results. Corpus search focuses on accuracy and only correct matches with 100% accuracy are regarded as results.

Many corpus search engines such as the IMS Corpus Workbench (CWB) and Poliqarp are based on a tabular data model whose rows represent token sequences and columns various annotations. CWB4 or Ziggurat attempts to overcome the limitations of CWB3 that can only handle up to 2.1 billion tokens and to meet all the CQLF metamodel levels by extending the CWB3 data model extensively (Evert and Hardie, 2015). However, to simplify the data model representation, access, and implementation, it would be limited to work only with static corpora where the tokenization and annotation values cannot be modified and the documents cannot be added to or deleted from the indexed corpora. Moreover, corpora cannot be combined into a single virtual corpus.

Relational databases are deemed to have problems to scale up to large corpora, particularly with billions of words (Evert and Hardie, 2015). Ghodke and Bird (2008)

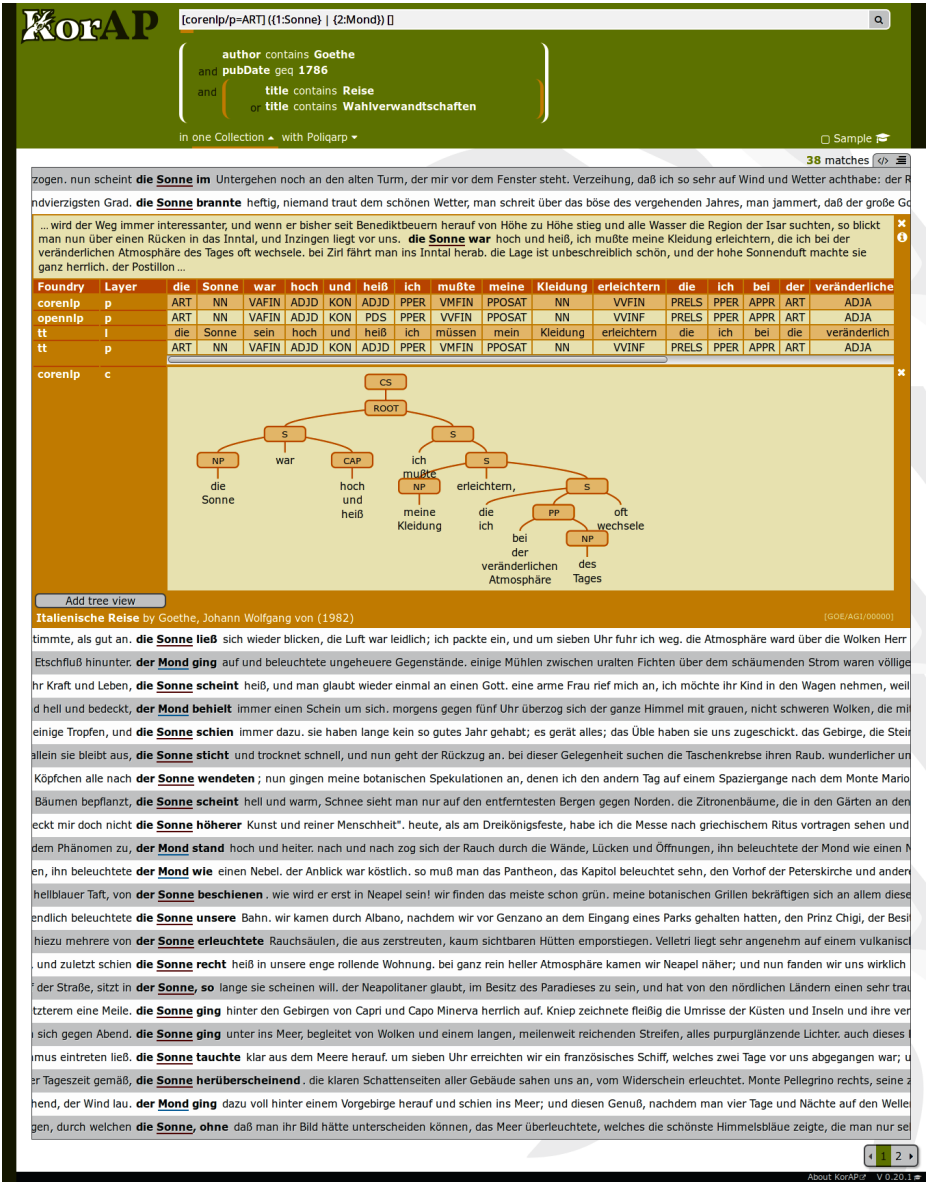

Figure 11: Screenshot of the Kalamar Frontend

illustrate this problem by comparing query execution time between a native XML DB (eXist) and a relational database (Oracle). Their experiments show that the query execution times of both approaches increase greatly as the dataset size increases. Recent evaluations have shown however, that parallelization can be a reasonable approach in dealing with these issues (Schneider, 2012). Besides, Davies (2005) suggests that his n-gram table approach has no limitations on the corpus size and the number of annotation tables and works very fast on pattern matching and synonym queries. However, Evert and Hardie (2015) argue about its redundancy issue and capability to handle more complex linguistic data structures. ANNIS (Zeldes et al., 2009) supports complex linguistic annotations, but it has only been tested with relatively small annotated corpora.[17]

XML database technologies such as BaseX and eXist rely heavily on XQuery and XPath to navigate and extract data from XML documents. Using XPath to query data with multiple stand-off annotation files is rather ineffective due to the need to resolve pointers between the files repeatedly (Mayo et al., 2006). The re-implementation of NXT's query language using XQuery (NQL, Mayo et al., 2006) is shown to be able to load about four times more data than the former NXT Search with Java implementation (Carletta et al., 2005) while utilizing the same memory size. However, using XQuery with stand-off data format decreases the speed performance in executing queries (Mayo et al., 2006).

Krill is based on prior research on the applicability of Lucene for complex linguistic corpus search tasks (Schnober, 2012). During the development of Krill, similar Lucene-based search engines emerged, like BlackLab[18] and MTAS.[19] BlackLab also supports multiple query languages (e.g. CQL, Lucene QL), but only basic features have been supported yet.[20] MTAS is a new corpus search engine in its early stages with support for distributed search using Apache Solr.[21]

7 Further Work

To support horizontal scalability in KorAP, multiple Krill instances can be utilized in a document-partitioned cluster for distributed search managed by Kanalito, that is still under development (see Sec. 2). With the introduction of this receptionist service, more features will be available, like facet search and methods for sorting, that have been already supported by COSMAS II. In addition, statistical analysis of query results through frequency and co-occurrence are in preparation.

We plan on evaluating Krill performance, for instance in terms of searching and indexing with regard to corpus size and in comparison to other corpus search systems,

[17] https://corpling.uis.georgetown.edu/annis-corpora/
[18] http://inl.github.io/BlackLab/
[19] https://meertensinstituut.github.io/mtas/
[20] https://github.com/INL/BlackLab/wiki/Blacklab-query-tool
[21] http://lucene.apache.org/solr/

such as COSMAS II. We also plan an evaluation of the scope of supported query constructs, especially with respect to the work on *CQLF* (Bański et al., 2016).

8 Summary

We have described Krill, a search component in KorAP, and its interaction with other components in the KorAP architecture. Krill is based on Lucene and uses an inverted index to perform full-text search. We have extended Lucene to support a wide range of corpus query operations on multiple annotations, and the flexible creation of virtual corpora. Krill is open source, available on GitHub[22] and published under the BSD-2 License. Despite being developed in the context of KorAP, Krill can be used as a stand-alone application.

9 Acknowledgements

We would like to thank our colleagues for their contributions, especially Carsten Schnober for his preliminary work on the KorAP search backend and Piotr Pęzik for his recommendations on the term structure. Regarding this manuscript we would like to thank all reviewers for their helpful feedback.

Krill is developed as part of the KorAP Corpus Analysis Platform at the Institute for the German Language, funded by the Leibniz-Gemeinschaft,[23] supported by the KobRA project,[24] funded by the Federal Ministry of Education and Research (BMBF), and the CLARIN-D project,[25] funded by the BMBF and the Ministry of Science, Research and the Arts (Baden-Württemberg).

References

Bański, P., Bingel, J., Diewald, N., Frick, E., Hanl, M., Kupietz, M., Pęzik, P., Schnober, C., and Witt, A. (2013a). KorAP: the new corpus analysis platform at IDS Mannheim. In Zygmunt Vetulani et al., editor, *Human Language Technologies as a Challenge for Computer Science and Linguistics. Proceedings of the 6th Language and Technology Conference*, pages 586–587, Fundacja Uniwersytetu im. A. Mickiewicza, Poznań, Poland.

Bański, P., Diewald, N., Hanl, M., Kupietz, M., and Witt, A. (2014). Access control by query rewriting: the case of KorAP. In Calzolari, N., Choukri, K., Declerck, T., Loftsson, H., Maegaard, B., Mariani, J., Moreno, A., Odijk, J., and Piperidis, S., editors, *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC 2014)*, pages 3817–3822, Reykjavik, Iceland. European Language Resources Association (ELRA).

Bański, P., Fischer, P. M., Frick, E., Ketzan, E., Kupietz, M., Schnober, C., Schonefeld, O., and Witt, A. (2012). The new IDS corpus analysis platform: Challenges and prospects. In

[22] https://github.com/KorAP/Krill

[23] http://www.leibniz-gemeinschaft.de/en/about-us/leibniz-competition/projekte-2011/2011-funding-line-2/

[24] http://www.kobra.tu-dortmund.de

[25] http://de.clarin.eu/

Calzolari, N., Choukri, K., Declerck, T., Doğan, M. U., Maegaard, B., Mariani, J., Odijk, J., and Piperidis, S., editors, *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC 2012)*, pages 2905–2911, Istanbul, Turkey. European Language Resources Association (ELRA).

Bański, P., Frick, E., Hanl, M., Kupietz, M., Schnober, C., and Witt, A. (2013b). Robust corpus architecture: a new look at virtual collections and data access. In Hardie, A. and Love, R., editors, *Corpus Linguistics 2013 Abstract Book*, pages 23–25, Lancaster. UCREL. `http://ucrel.lancs.ac.uk/cl2013/doc/CL2013-ABSTRACT-BOOK.pdf`.

Bański, P., Frick, E., and Witt, A. (2016). Corpus Query Lingua Franca (CQLF). In Calzolari, N., Choukri, K., Declerck, T., Goggi, S., Grobelnik, M., Maegaard, B., Mariani, J., Mazo, H., Moreno, A., Odijk, J., and Piperidis, S., editors, *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016)*, pages 2804–2809, Portorož, Slovenia. European Language Resources Association (ELRA).

Bingel, J. and Diewald, N. (2015). KoralQuery – a general corpus query protocol. In *Proceedings of the Workshop on Innovative Corpus Query and Visualization Tools at NODALIDA 2015*, Vilnius, Lithuania.

Bodmer, F. (1996). Aspekte der Abfragekomponononente von COSMAS-II. *LDV-INFO. Informationsschrift der Arbeitsstelle Linguistische Datenverarbeitung*, 8:112–122.

Büttcher, S., Clarke, C. L. A., and Cormack, G. V. (2010). *Information Retrieval: Implementing and Evaluating Search Engines*. MIT Press.

Carletta, J., Evert, S., Heid, U., and Kilgour, J. (2005). The NITE XML Toolkit: data model and query. *Language Resources and Evaluation Journal*, 39(4):313–334.

Chambi, S., Lemire, D., Kaser, O., and Godin, R. (2016). Better bitmap performance with Roaring bitmaps. *Software: Practice and Experience*, 46(5):709–719. `http://arxiv.org/abs/1402.6407`.

Davies, M. (2005). The advantage of using relational databases for large corpora: Speed, advanced queries and unlimited annotation. *International Journal of Corpus Linguistics*, 10(3):307–334.

Diewald, N. and Bingel, J. (2015). KoralQuery 0.3. Technical report, IDS Mannheim, Germany. Working draft, `https://KorAP.github.io/Koral`, last accessed 2/2/2017.

Diewald, N., Hanl, M., Margaretha, E., Bingel, J., Kupietz, M., Bański, P., and Witt, A. (2016). KorAP architecture – diving in the deep sea of corpus data. In Calzolari, N., Choukri, K., Declerck, T., Goggi, S., Grobelnik, M., Maegaard, B., Mariani, J., Mazo, H., Moreno, A., Odijk, J., and Piperidis, S., editors, *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016)*, pages 3586–3591, Portorož, Slovenia. European Language Resources Association (ELRA).

Evert, S. and Hardie, A. (2015). Ziggurat: A new data model and indexing format for large annotated text corpora. In *Proceedings of the 3rd Workshop on Challenges in the Management of Large Corpora (CMLC-3)*, pages 21–27.

Evert, S. and the OCWB Development Team (2010). CQP query language tutorial. Technical report, The OCWB Development Team. `http://cwb.sourceforge.net/files/CQP_Tutorial.pdf`.

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

Kupietz, M. and Lüngen, H. (2014). Recent developments in DeReKo. In Calzolari, N., Choukri, K., Declerck, T., Loftsson, H., Maegaard, B., Mariani, J., Moreno, A., Odijk, J., and Piperidis, S., editors, *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC 2014)*, pages 2378–2385, Reykjavik, Iceland. European Language Resources Association (ELRA).

Lüngen, H. and Sperberg-McQueen, C. M. (2012). A TEI P5 document grammar for the IDS text model. *Journal of the Text Encoding Initiative*, 3. `http://jtei.revues.org/508`.

Mayo, N., Kilgour, J., and Carletta, J. (2006). Towards an alternative implementation of NXT's query language via XQuery. In *Proceedings of the 5th Workshop on NLP and XML (NLPXML-2006)*, pages 27–34, Trento, Italy.

Munro, J. I., Papadakis, T., and Sedgewick, R. (1992). Deterministic skip lists. In *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '92, pages 367–375, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Przepiórkowski, A., Krynicki, Z., Dębowski, Ł., Woliński, M., Janus, D., and Bański, P. (2004). A search tool for corpora with positional tagsets and ambiguities. In *Proceedings of the Fourth International Conference on Language Resources and Evaluation, LREC 2004*, pages 1235–1238.

Pugh, W. (1990). Skip Lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676.

Rosenfeld, V. (2010). An implementation of the Annis 2 query language. Technical report, Humboldt-Universität zu Berlin.

Schindler, U. and Diepenbroek, M. (2008). Generic XML-based framework for metadata portals. *Computers & Geosciences*, 34(12):1947–1955. `http://epic.awi.de/17813/1/Sch2007br.pdf`.

Schneider, R. (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).

Schnober, C. (2012). Using information retrieval technology for a corpus analysis platform. In *Proceedings of Konvens 2012*, pages 199–207, Vienna, Austria.

Sporny, M., Longley, D., Kellogg, G., Lanthaler, M., and Lindström, N. (2014). JSON-LD 1.0. a JSON-based serialization for linked data. Technical report, W3C. W3C Recommendation; `http://www.w3.org/TR/json-ld/`.

Zeldes, A., Ritz, J., Lüdeling, A., and Chiarcos, C. (2009). ANNIS: A search tool for multi-layer annotated corpora. In Mahlberg, M., González-Díaz, V., and Smith, C., editors, *Proceedings of the Corpus Linguistics 2009 Conference*, Liverpool, UK.

Zobel, J. and Moffat, A. (2006). Inverted files for text search engines. *ACM Computing Surveys*, 38(2).