KoralQuery – a General Corpus Query Protocol

Joachim Bingel, Nils Diewald
Institut für Deutsche Sprache
Mannheim, Germany
bingel,diewald@ids-mannheim.de

Abstract

The task-oriented and format-driven development of corpus query systems has led to the creation of numerous corpus query languages (QLs) that vary strongly in expressiveness and syntax. This is a severe impediment for the interoperability of corpus analysis systems, which lack a common protocol. In this paper, we present KoralQuery, a JSON-LD based general corpus query protocol, aiming to be independent of particular QLs, tasks and corpus formats. In addition to describing the system of types and operations that KoralQuery is built on, we exemplify the representation of corpus queries in the serialized format and illustrate use cases in the KorAP project.

1 Introduction

In the past, several corpus query systems have been developed with the purpose of exploring and providing access to text corpora, often under the assumption of specific linguistic questions that the annotated corpora have been expected to help answer. This task-oriented and format-driven development has led to the creation of several distinct corpus query languages (QLs), including those mentioned in Section 3. Such QLs vary strongly in expressiveness and usability (Frick et al., 2012).

This brings several unpleasant consequences both for researchers and developers. For instance, the researcher who uses a particular system must formulate her queries in no other QL than the one used for this system, which might require additional training prior to the actual research. It might even be the case that certain research questions cannot be answered due to limitations of the QL, while the actual query system and the underlying corpus data could in fact provide results. For developers, the lack of a common protocol prevents interoperability between different query systems, for instance to forward user requests from one system to another, which may have access to additional resources.

In this paper, we present KoralQuery, a general protocol for the representation of requests to corpus query systems independent of a particular query language. KoralQuery provides an extensible system of different linguistic and metalinguistic types and operations, which can be combined to represent queries of great complexity. Several query languages can thus be mapped to a common representation, which lets users of query systems formulate queries in any of the QLs for which such a mapping is implemented (cf. Section 4). Further benefits of KoralQuery include the dynamic definition of virtual corpora and the possibility to simultaneously access several, concurrent layers of annotation on the same primary textual data.

2 Related Work

In former publications, KoralQuery was introduced as a unified serialization format for CQLF[1] (Bański et al., 2014), a companion effort focussing on the identification and theoretical description of corpus query concepts and features.

Another approach to a common query language that is independent of tasks and formats is CQL (*Contextual Query Language*) (OASIS Standard, 2013), with its XML serialization format XCQL.[2] KoralQuery differs from CQL in focussing on queries of linguistic structures, and separating document and span query concepts (see Section 3).

[1] CQLF is short for *Corpus Query Lingua Franca*, which is part of the ISO TC37 SC4 Working Group 6 (ISO/WD 24623-1).

[2] Like KoralQuery, XCQL is not meant to be human readable, but to represent query expressions as machine readable tree structures. For various compilers from CQL to XCQL, see http://zing.z3950.org/cql/; last accessed 27 April 2015.

3 Query Representation

KoralQuery is serialized to JSON-LD (Sporny et al., 2014), a JSON (Crockford, 2006) based format for Linked Data, which makes it possible for corpus query systems to interoperate by exchanging the common protocol.[3] JSON-LD relies on the definition of object types via the `@type` keyword, thus informing processing software of the attributes and values that a particular object may hold. As can be seen in the example serializations in this section (see Fig. 1-3), KoralQuery makes use of the `@type` keyword to declare query object types. Those types fall into different categories that we introduce in the remainder of this section.[4]

While KoralQuery aims to express as many different linguistic and metalinguistic query structures as possible, it currently guarantees to represent types and operations defined in *Poliqarp QL* (Przepiórkowski et al., 2004), *COSMAS II QL* (Bodmer, 1996) and *ANNIS QL* (Rosenfeld, 2010). In addition, the protocol comprises a subset of the elements of CQL (OASIS Standard, 2013).

As JSON-LD objects can reference further namespaces (via the `@context` attribute), KoralQuery is arbitrarily extensible.

3.1 Document Queries

KoralQuery allows to specify metadata constraints that act as filters for virtual collections using the `collection` attribute. Those metadata constraints, so-called **collection types**, serve a dual purpose: Besides the obvious benefit of allowing users to restrict their search via dynamic sampling to documents that meet specific requirements on metadata such as publication date, authorship or genre, they can be used to control access to texts that the user has no permission to read (cf. Sec. 3.3).

A single metadata constraint is called a **basic collection type**, and defines a metadata field, a value and a match modifier, for example to negate the constraint. Basic collection types can be combined using boolean operators (AND and OR) to recursively form **complex collection types**. The result of a collection type is a collection of documents which satisfy the encoded constraint (or combination of constraints), for instance all documents that were published after a certain date or that contain a certain string of characters in their title. Figure 1 illustrates the serialization of a simple virtual collection definition.

```
{
  "@context" : "http://korap.ids-mannheim.de/ns/
      koral/0.3/context.jsonld",
  "collection" : {
    "@type" : "koral:doc",
    "key" : "pubDate",
    "value" : "2005-05-25",
    "type" : "type:date",
    "match" : "match:geq"
  },
  "query" : {}
}
```

Figure 1: KoralQuery serialization for a virtual collection that is restricted to documents with a `pubDate` of greater or equal than `2005-05-25`.

3.2 Span Queries

To find occurrences of particular linguistic structures in corpus data (possibly restricted through the aforementioned document queries), KoralQuery uses the attribute `query`, under which it registers objects of specific, well-defined types. Those objects, along with their hierarchical organization, represent the linguistic query issued by the user.[5]

The intended generic usability of KoralQuery demands a high degree of flexibility in order to cover as many linguistic phenomena and theories as possible. It must therefore be maximally independent of, and neutral with regard to,

(i) the type and structure of linguistic annotation on the text data,
(ii) the choice of specific tag sets, e.g. for part-of-speech annotations or dependency labels.

KoralQuery achieves this neutrality by instantiating distinct linguistic types as abstract structures which can flexibly address different sources and layers of linguistic annotation at the same time. Linguistic patterns of greater complexity can be defined by using a modular system of nestable types and operations, drawing on various familiar search technologies and formalisms, includ-

[3] JSON-LD was chosen to be compatible with LAPPS recommendations from ISO TC37 SC4 WG1-EP, as suggested by Piotr Bański.

[4] The type categories are set in boldface. A detailed definition of types and attributes is provided by the KoralQuery specification (Diewald and Bingel, 2015), which may serve as a reference for implementers of KoralQuery processors.

[5] As the response format is not part of the KoralQuery specification, the result handling is subject to the query engine. It may, for instance, return surrounding text spans or the total number of occurrences.

ing concepts from regular expressions, XML tree traversal, boolean search and relational database queries.

The nesting principle of KoralQuery states that objects describing linguistic structures in the corpus data, so-called **span types**, may be embedded in parental objects to recursively describe complex linguistic structures, thus forming a single-rooted tree.

Span types may be further sub-classified into basic and complex types. **Basic span types** denote linguistic entities such as words, phrases and sentences that are annotated in the corpus data. The result of such a span type is a text span, which in turn is defined through a start and an end offset with respect to the primary text data. **Complex span types** define linguistic or result-modifying operations on a set of embedded span types, which thus act as arguments (or *operands*) of the relation and pass their resulting text spans on to the parent operation.[6] Such operations may express syntactic relations or positional constraints between spans.

Figure 2, for example, represents a span query of two `koral:token` objects (basic span types) each wrapping a single `koral:term` object, whose resulting text spans are required to be in a sequence (i.e. follow each other immediately in the order they appear in the list), as formulated by the `operation:sequence` in the embedding `koral:group` object (a complex span type).

Leaf objects of the span query tree structure may either be basic span types or **parametric types**, containing specific information that is requested for certain span types. They are intended to normalize the usage and representation of similar or equal parameters used across different types. The `koral:term` objects in Figure 2, which express constraints on their parent `koral:token` objects, are examples of such parametric types and are used to uniformly access annotation labels from different sources and on different layers. Next to such **basic parametric types**, KoralQuery provides **complex parametric types** that encode, for instance, logical operations on other parametric types (see the `koral:termGroup` in Figure 2).

Note that all of those types are themselves complex structures in that they are composed of a specific set of obligatory and optional attributes that carry corresponding values. Those values, in turn, are also constrained to be of specific data types. They can either be primitives (like string, integer or boolean), parametric KoralQuery types, or controlled values.

[6] In addition, the `koral:reference` type may refer to objects elsewhere in the tree, which provides a mechanism similar to ID/IDREF in XML. This strategy is necessary to support graph-based query structures found in certain query languages.

```
{
  "@context" : "http://korap.ids-mannheim.de/ns/
      koral/0.3/context.jsonld",
  "collection" : {},
  "query" : {
    "@type":"koral:group",
    "operation" : "operation:sequence",
    "operands" : [ {
      "@type" : "koral:token",
      "wrap" : {
        "@type" : "koral:termGroup",
        "relation" : "relation:and",
        "operands" : [ {
          "@type" : "koral:term",
          "foundry" : "tt",
          "key" : "ADJA",
          "layer" : "pos",
          "match" : "match:eq"
        }, {
          "@type" : "koral:term",
          "foundry" : "cnx",
          "key" : "@PREMOD",
          "layer" : "syn",
          "match" : "match:eq"
        } ]
    }, {
      "@type" : "koral:token",
      "wrap" : {
        "@type" : "koral:term",
        "key" : "octopus",
        "layer" : "lemma",
        "match" : "match:eq"
      }
    } ]
  }
}
```

Figure 2: KoralQuery serialization for a premodifying adjective followed by the lemma *octopus*. The dual constraint on the first token (adjective and premodifying) is reflected by the `koral:termGroup`, which expresses a conjunction of the two `koral:term` objects. The different values for `foundry` indicate that different annotation sources are addressed.

3.3 Query Rewrites

Query processors may perform a wide range of different tasks aside of searching. Examples include the modification of queries to restrict access to certain documents, to improve recall (e.g. by introducing synonyms or suggesting query reformulations), or to inject missing query elements (like

```
{
  "@context" : "http://korap.ids-mannheim.de/ns/
      koral/0.3/context.jsonld",
  "collection" : {
    "@type" : "koral:docGroup",
    "operation" : "operation:and",
    "operands" : [ {
      "@type" : "koral:doc",
      "key" : "pubDate",
      "value" : "2005-05-25",
      "type" : "type:date",
      "match" : "match:geq"
    }, {
      "@type" : "koral:doc",
      "key" : "corpusID",
      "value" : "Wikipedia",
      "rewrites" : [ {
        "@type" : "koral:rewrite",
        "src" : "Kustvakt",
        "operation" : "operation:injection"
      } ]
    } ]
  },
  "query" : {}
}
```

Figure 3: Rewritten KoralQuery instance (see Figure 1), with an injected access restriction.

preferred annotation tools) based on user settings (Bański et al., 2014). Queries may also be analyzed for the most commonly queried structures, for instance to perform query and index optimization or to shed light on which texts and annotations are most popular with the users. In a post-processing step, queries can also be transformed for visualization purposes, for example to illustrate sequences or alternatives in complex query structures.

Using a well-defined and widely adopted serialization format such as JSON makes it easy to perform such tasks, and KoralQuery supports this kind of pre- and post-processors even further by introducing mechanisms to trace query rewrites by using so-called **report types** that are passed to further processors in the processing pipeline. In this way, query modifications (like the aforementioned rewrites for access restriction and recall improvements) can be made visible and transparent to the user. In this respect, KoralQuery differs from common database query systems, where rewrites are internal and hidden from the user (Huey, 2014).

In Figure 3, the virtual collection of Figure 1 is rewritten by the processor *Kustvakt* in a way that a further constraint is injected, limiting the virtual collection to all documents with a `corpusID` of `Wikipedia` (i.e. excluding all documents from other corpora). This rewrite is documented by the `koral:rewrite` object (a report type). Documenting rewrites is optional (e.g. the injected `operation:and` in the example Figure is implicit and was not reported using `koral:rewrite`).

In addition, KoralQuery allows to report on various processing issues (independent of rewrites, e.g. regarding incompatibilities) by using the `errors`, `warnings`, and `messages` attributes.

Report types (in opposition to collection types, span types, and parametric types) do not alter the expected query result.

4 Implementations

KoralQuery is the core protocol used in KorAP[7] (Bański et al., 2013), a corpus analysis platform developed at the Institute for the German Language (IDS). KorAP is designed to handle very large corpora and to be sustainable with regard to future developments in corpus linguistic research. This is ensured through a modular architecture of interoperating software units that are easy to maintain, extend and replace. The interoperability of components in KorAP is certified through the use of KoralQuery for all internal communications.

Koral[8] translates queries from various corpus query languages (as mentioned in Section 3) to corresponding KoralQuery documents. This conversion is a two-stage process, which first parses the input query string using a context-free grammar and the ANTLR framework (Parr and Quong, 1995) before it translates the resulting parse tree to KoralQuery.

Krill[9] is a corpus search engine that expects KoralQuery instances as a request format. To index and retrieve primary data, textual annotations and metadata of documents as formulated by KoralQuery, Krill utilizes *Apache Lucene*.[10]

Kustvakt is a user and corpus policy management service that accepts KoralQuery requests and rewrites the query as a preprocessor (see Sec. 3.3) before it is passed to the search engine (e.g. Krill). Rewrites of the document query may restrict the requested collection to documents the user is allowed to access, while the span query may be modified by injecting user defined properties.

[7] `http://korap.ids-mannheim.de/`

[8] `http://github.com/KorAP/Koral`; Koral is free software, licensed under BSD-2.

[9] `http://github.com/KorAP/Krill`; Krill is free software, licensed under BSD-2.

[10] `http://lucene.apache.org/core/`

5 Summary and Further Work

We have presented KoralQuery, a general protocol for queries to linguistic corpora, which is serialized as JSON-LD. KoralQuery allows for a flexible representation and modification of corpus queries that is independent of pre-defined tag sets or annotation schemes. Those queries pertain to both selection of documents by metadata or content, and text span retrieval by the specification of linguistic patterns. To this end, the protocol defines a set of types and operations which can be nested to express complex linguistic structures. By employing an automatic conversion from several QLs to KoralQuery, corpus engines may allow their users to choose the QL that they are most comfortable with or that are best equipped to answer their research questions.

The KoralQuery specification (Diewald and Bingel, 2015) does not claim to be complete or to cover all possible linguistic types and structures. Amendments to the protocol may follow in future versions or may be implemented by individual projects, which is easily done by supplying an additional JSON-LD `@context` file that links new concepts to unique identifiers. Extensions that we consider for upcoming versions of KoralQuery include text string queries that are not constrained by token boundaries and more powerful stratification techniques for virtual collections.

Acknowledgements

KoralQuery, as well as the described implementation components, are developed as part of the KorAP project at the Institute for the German Language (IDS)[11] in Mannheim, member of the Leibniz-Gemeinschaft, and supported by the KobRA[12] project, funded by the Federal Ministry of Education and Research (BMBF), Germany. The authors would like to thank their colleagues for their valuable input.

References

Piotr Bański, Joachim Bingel, Nils Diewald, Elena Frick, Michael Hanl, Marc Kupietz, Piotr Pezik, Carsten Schnober, and Andreas Witt. 2013. KorAP: the new corpus analysis platform at IDS Mannheim. In Zygmunt Vetulani and Hans Uszkoreit, editors, *Human Language Technologies as a Challenge for Computer Science and Linguistics. Proceedings of the 6th Language and Technology Conference*, Poznań. Fundacja Uniwersytetu im. A. Mickiewicza.

Piotr Bański, Nils Diewald, Michael Hanl, Marc Kupietz, and Andreas Witt. 2014. Access Control by Query Rewriting: the Case of KorAP. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC 2014)*, Reykjavik, Iceland, may. European Language Resources Association (ELRA).

Franck Bodmer. 1996. Aspekte der Abfragekomponente von COSMAS II. *LDV-INFO*, 8:142–155.

Douglas Crockford. 2006. The application/json Media Type for JavaScript Object Notation (JSON). Technical report, IETF, July. `http://www.ietf.org/rfc/rfc4627.txt`.

Nils Diewald and Joachim Bingel. 2015. KoralQuery 0.3. Technical report, IDS, Mannheim, Germany. Working draft, in preparation, `http://KorAP.github.io/Koral`, last accessed 27 April 2015.

Elena Frick, Carsten Schnober, and Piotr Bański. 2012. Evaluating query languages for a corpus processing system. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC 2012)*, pages 2286–2294.

Patricia Huey, 2014. *Oracle Database, Security Guide, 11g Release 1 (11.1)*, chapter 7. Using Oracle Virtual Private Database to Control Data Access, pages 233–272. Oracle. `http://docs.oracle.com/cd/B28359_01/network.111/b28531.pdf`, last accessed 27 April 2015.

OASIS Standard. 2013. searchRetrieve: Part 5. CQL: The Contextual Query Language Version 1.0. `http://docs.oasis-open.org/search-ws/searchRetrieve/v1.0/os/part5-cql/searchRetrieve-v1.0-os-part5-cql.html`.

Terence J. Parr and Russell W. Quong. 1995. ANTLR: A predicated-LL (k) parser generator. *Software: Practice and Experience*, 25(7):789–810.

Adam Przepiórkowski, Zygmunt Krynicki, Lukasz Debowski, Marcin Wolinski, Daniel Janus, and Piotr Bański. 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. European Language Resources Association (ELRA).

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

Manu Sporny, Dave Longley, Gregg Kellogg, Markus Lanthaler, and Niklas Lindström. 2014. JSON-LD 1.0 – A JSON-based Serialization for Linked Data. Technical report, W3C. W3C Recommendation, `http://www.w3.org/TR/json-ld/`.

[11] `http://ids-mannheim.de/`

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