Proceedings of he 8th Workshop on Challenges in the Management of Large Corpora, pages 31–39
Language Resources and Evaluation Conference (LREC 2020), Marseille, 11–16 May 2020

The Corpus Query Middleware of Tomorrow – A Proposal for a Hybrid Corpus Query Architecture

Markus Gärtner
Institute for Natural Language Processing
University of Stuttgart
markus.gaertner@ims.uni-stuttgart.de

Abstract

Development of dozens of specialized corpus query systems and languages over the past decades has let to a diverse but also fragmented landscape. Today we are faced with a plethora of query tools that each provide unique features, but which are also not interoperable and often rely on very specific database back-ends or formats for storage. This severely hampers usability both for end users that want to query different corpora and also for corpus designers that wish to provide users with an interface for querying and exploration. We propose a hybrid corpus query architecture as a first step to overcoming this issue. It takes the form of a middleware system between user front-ends and optional database or text indexing solutions as back-ends. At its core is a custom query evaluation engine for index-less processing of corpus queries. With a flexible JSON-LD query protocol the approach allows communication with back-end systems to partially solve queries and offset some of the performance penalties imposed by the custom evaluation engine. This paper outlines the details of our first draft of aforementioned architecture.

Keywords: corpus query system, query language, middleware

1. Introduction

For roughly 30 years specialized corpus query systems (CQSs) have aided researchers in the exploration or evaluation of corpora. During all this time a plethora of different implementations and architectures emerged, with distinctive features or specialization for particular use cases. As corpus resources grow steadily in both size (the number of primary segments such as tokens) and complexity (the number and type of interrelated annotation layers), the need for dedicated query interfaces also became more pronounced.

However, especially the latest generation of CQSs developed during the last decade has revealed an overall decline of expressiveness in their query languages compared to the peak era prior to it. While earlier systems or languages such as FSQ (Kepser, 2003), MonaSearch (Maryns and Kepser, 2009) or LPath^{+} (Lai and Bird, 2005) offered pretty much the full expressive power of first-order logic, later instances have mostly been limited to the existential fragment (such as ANNIS3 (Krause and Zeldes, 2014), ICARUS (Gärtner et al., 2013) or SETS (Luotolahti et al., 2015)) with only limited support for quantification, negation or closures to cope with the performance issues caused by evaluating complex queries on increasingly larger corpora[1].

Besides obvious scalability reasons, the expressiveness of a CQS can also be a direct result of architectural choices, especially the monolithic approach common to many query engines:

Many CQSs today builds on general purpose database solutions (such as relational database management systems (RDBMSs)) or text indexing frameworks and subsequently delegate the actual query evaluation to this back-end system by translating the original user query from the respective corpus query language (CQL). As such the entire software stack and typically also the CQL itself are bound (and limited) to the data model of this back-end system, giving rise to a series of recurring issues. If a query constraint cannot be directly expressed or evaluated in the underlying (database) query language, it typically won't be available in the CQL (Section 3 lists certain exceptions from this trend). Similarly, if a feature or phenomenon is not explicitly encoded in the back-end storage, it often cannot be used for querying. Last but not least the handling of query results (eloquently dubbed the "Achilles heel of corpus query tools" by Mueller (2010)) differs greatly between systems. From flat or keyword-in-context view in COSMAS (Bodmer, 2005) or FSQ to elaborate tree visualizations in ANNIS or ICARUS, CQSs offer a wide variety of result formats or visual result inspection interfaces, but individual solutions are usually limited to a small subset of this diversity.

To overcome these limitations we propose a novel approach for a hybrid corpus query architecture that combines the performance benefits of modern database and text indexing systems with the flexibility of a custom query evaluation engine[2]. It takes the form of a middleware system called ICARUS2 Query Processor (IQP) between query front-ends and corpus storage back-ends. Due to its modular approach it could also serve as a platform for unification between the heterogeneous tangle of corpus query languages. As this is work in progress we mainly intend to sketch the outline of the overall architecture and its technical details, and open up its merits for discussion with other experts in the field.

The remaining part of this paper is structured as follows: We introduce the overall goals and (preliminary) limitations of our approach in Section 2 and contextualize it within the state of the art in Section 3. Section 4 provides an in-depth

[1]Cf. (Kepser, 2004) in the context of FSQ.

[2]We use this term as a substitution for query engines that can perform the entire query evaluation themselves, typically in-memory on a live corpus and index-less.

overview of the different components in the architecture, including example queries to highlight query capabilities. Section 5 contains location and licensing information for the source code and Section 6 concludes.

2. Goals and Limitations

With IQP we aim at solving a rather broad range of issues. The primary goals of our proposed architecture are the following, with certain initial limitations listed afterwards:

1. To provide **unified query endpoint**. That is, a query language and associated protocol for expressing arbitrary information needs against linguistic corpora. This does however not pertain to any form of user interface, as the entire system as described in Section 4 is meant to be embedded as a middleware within an outer infrastructure that provides the graphical means for user interactions.
2. Implementation of a **custom query engine** in the form of a modular, extensible and scalable evaluation engine for queries provided by aforementioned protocol. As hinted at in Section 4.3 the evaluation complexity for queries can easily become exponential in the size of whatever the unit-of-interest (UoI) for the query is. It is therefore difficult to make general performance guarantees and we estimate the overall performance to be several magnitudes behind specialized index-based alternatives. While this might sound prohibitive for large-scale usage, it should be considered a small price to pay for the availability of extended query options.
3. Interfaces to **optional back-ends** to maximally exploit the performance benefits of existing database and test indexing systems. Ideally this can be realized in a black-box design where the middleware itself only needs to be concerned with a series of (service) endpoints to whom preprocessed queries can be forwarded and which return result candidates and information about solved query sub-parts.

While the architecture sketch in Section 4 displays the entire query evaluation workflow, there are a few components and aspects that we do not intend to fully address in the first prototype phase of our middleware, leading to a few (temporary) limitations on the following aspects:

Result preparation While ultimately of great importance in the long run, we initially focus on the query evaluation itself and leave the extended result processing for a later iteration. The query protocol in Section 4.2 contains placeholders for the subsequent declaration of result formats and script-like processing instructions, but basic result settings such as size, sorting or filtering are already part of the initial draft.

Back-end wrappers As back-end systems are optional and the evaluation engine is expected to be able to handle queries without external help, we do not plan to include actual wrappers for back-ends in the early development.

Graph evaluation While the specification for our query language includes structural constraints for graph constructs (cf. Section 4.3.2), the engine will only be able to evaluate sequence and tree constraints in the first prototype, as those two types also correspond to the predominant data structures used in corpus modeling[3].

3. Related Work

For a very detailed overview of existing CQL families and types of CQSs we refer to the recent work of Clematide (2015). In the remainder of this section we only highlight those (types of) CQSs that are most relevant to our proposed approach or which implement a similar concept.

3.1. Custom Query Engines

The concept of implementing a custom query engine is not entirely new. In fact, several successful CQS already feature their very own evaluation engines:
TIGERSearch (König and Lezius, 2000; Lezius, 2002), FSQ and ICARUS all ship with a query engine that can match structural queries in-memory against a treebank. Similarly, PML-TQ (Pajas and Štěpánek, 2009; Štěpánek and Pajas, 2010) allows to switch its RDBMS back-end for an integrated index-less evaluator implemented in Perl, turning it into a custom query engine for local data.
The popular Corpus Workbench with its Corpus Query Processor (CQP) (Christ, 1994; Evert and Hardie, 2011) is representative for the family of CQSs that provide a custom query engine but at the same time also rely on their own indexing to preprocess corpus data in order to improve query performance. As the associated CQLs of the newer systems mentioned above remain quite limited[4] in their expressiveness compared to our proposed ICARUS2 Query Language (IQL) in Section 4.3, we treat those CQSs as equivalent to off-the-shelf database or text indexing solutions for the purpose of our approach.

3.2. Hybrid Solutions

While traditionally many CQSs implemented the *query translation* approach[5], several systems go beyond that and employ a hybrid strategy for query evaluation.
SETS (Luotolahti et al., 2015) and TreeAligner (Lundborg et al., 2007; Marek et al., 2008) build on RDBMSs for storage, but complement it with their own query evaluation. Slightly different, Ghodke and Bird (2012) extend the text indexing and query engine LUCENE[6] with a custom indexing scheme for storing treebank information.
Those approaches lack a broad coverage wrt query expressiveness, but serve as show cases for successfully evaluating very specific (treebank) queries in a highly scalable manner. Consequently, they too are prime candidates for back-ends in the architecture described in Section 4.

[3]While some approaches, such as SALT (the model behind ANNIS), successfully model complete corpora entirely as graphs, the individual components like sentences or syntax annotations naturally form sequence or tree structures.

[4]The PML-TQ system does however offer the most flexible result processing interface we are aware of, which definitely is an inspiring baseline for the future design of a component in IQP with similar roles.

[5]Using general purpose database or indexing solutions to store the corpus data and delegate the entire query evaluation to this back-end by translating it into its native query language.

[6]`https://lucene.apache.org/`

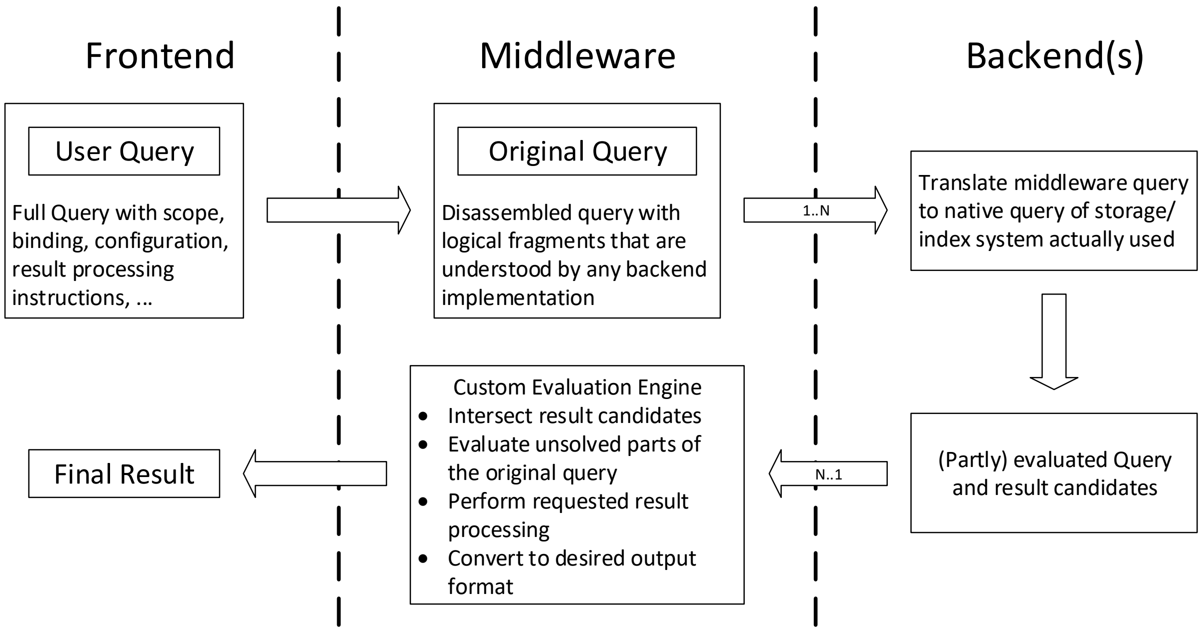
Frontend
Middleware
Backend(s)
User Query
Full Query with scope, binding, configuration, result processing instructions, …
Original Query
Disassembled query with logical fragments that are understood by any backend implementation
1..N
Translate middleware query to native query of storage/ index system actually used
Custom Evaluation Engine
Intersect result candidates
Evaluate unsolved parts of the original query
Perform requested result processing
Convert to desired output format
Final Result
N..1
(Partly) evaluated Query and result candidates

Figure 1: Sketch of the architecture service components and the query data flow.

3.3. Unification & Standardization

Existing efforts on unification or standardization of corpus querying typically focus on the query language level, with notable examples being CQLF[7] (Bański et al., 2016) for a general standardization initiative and KorAP (Bański et al., 2014; Diewald et al., 2016) and CLARIN FCS [8] as actual implementations.
KorAP lets the user choose upfront among several corpora and supported query languages to express a query and then does a N:1 translation into KoralQuery (Bingel and Diewald, 2015), a JSON-LD based query protocol and instantiation of CQLF. The KoralQuery expression is subsequently evaluated by the back-end responsible for the chosen corpus, typically by yet another translation into the respective query language (for the relational database (RDB), LUCENE index or the graph database Neo4j[9]).
On the other hand, the front end of CLARIN FCS provides a single corpus query language with essentially a subset of the expressiveness offered by the individual endpoints that can be queried with it. FCS queries are distributed to different endpoints that participate in the search framework, evaluated locally on each node (akin to a global 1:N translation) and the individual results are then aggregated centrally again and presented to the user.
Both of these approaches improve considerably upon the clutter of (incompatible) corpus query languages and/or evaluation systems. But they also introduce or maintain strong couplings between the front-end (expressed by the CQL or set of CQLs) and the back-end (defined by the actual evaluation engine, typically a RDBMS or similar), and in doing so render it quite difficult to make substantial changes to either.
In the following section we present an architecture that partly resembles KorAP, but introduces an additional fully-independent custom evaluation engine together with a query protocol that completely decouples the traditional front- and back-end roles.

4. Hybrid Architecture

This section describes the proposed architecture for a hybrid corpus query system and its most important components in detail. The overall concept is to decouple features of potential back-end systems and front-end concerns or query language properties. We achieve this by a dedicated middleware layer that mediates between front- and back-end instances and fills feature gaps when it comes to query evaluation or result preparation. Fig. 1 shows this middleware embedded in a typical scenario with a front-end for query construction and result presentation and (optional) back-end(s) with specialized storage and indexing capabilities that allow efficient evaluation of certain query constraints.

4.1. Data Model

IQP build on the ICARUS2 Corpus Modeling Framework (ICMF) (Gärtner and Kuhn, 2018) as its interface to interacting with corpus data. ICMF applies the concept *separation of concerns* to varies aspects of corpus modeling:
Each corpus resource is required to be accompanied by a set of metadata describing its composition, dependencies on other resources, how to access it and optionally providing details on tagsets or other annotation-related information. The data model of ICMF organizes corpora as hier-

[7]**C**orpus **Q**uery **L**ingua **F**ranca. Part of the ISO TC37 SC4 Working Group 6 (ISO/WD 24623-1).
[8]`https://www.clarin.eu/content/content-search`
[9]`http://neo4j.com/`

```
{ "@context" : "http://www.ims.uni-stuttgart.de/icarus/v2/jsonld/iql/query"
"@type" : "iql:Query",
"iql:imports" : [ {
        "@type" : "iql:Import",
        "iql:name" : "common.tagsets.stts"
} ],
"iql:setup" : [ {
        "@type" : "iql:Property",
        "iql:key" : "iql.string.case.off"
} ],
"iql:streams" : [ {
        "@type" : "iql:Stream",
        "iql:corpus" : {
                "@type" : "iql:Corpus",
                "iql:name" : "TIGER v2"
        },
        "iql:rawPayload" : "FIND ADJACENT [pos==stts.ADJ][form==\"test\"]",
        "iql:result" : {
                "@type" : "iql:Result",
                "iql:resultTypes" : [ "kwic" ],
                "iql:limit" : 1000
        }
} ] }
```

Figure 2: A simple mock-up query to illustrate the JSON-LD representation used in the protocol described in Section 4.2. The query searches the TIGER corpus for word pairs that start with an adjective and end in a word with the surface form "test", ignoring case and only returning up to 1000 hits in a KWIC style. The query also features an import statement for the STTS part-of-speech tagset, allowing more controlled expressions inside query constraints.

archical collections of inter-related layers, each with definite responsibilities. One family of layers is strictly used to model the logical and structural composition of a corpus, such as segmentation, hierarchical grouping and relational structures, such as syntax, discourse or coreference in a text corpus. Separated from structural concerns, the actual content (e.g. text, audio or linguistic annotations) is modeled by another layer type acting as mapping from corpus elements to their respective annotations or text content.
This separation of concerns allows the evaluation engine in IQP to also efficiently separate certain aspects of queries and query evaluation: The metadata level provides the basis for binding variables in a query to actual objects in the corpus. It also helps restricting the methods and properties available to expressions inside structural constraints (cf. Section 4.3.2), depending on whether they represent mere sequential structures (such as sentences) or more complex data types, for instance syntax trees. Evaluation of the latter also only needs access to structure-related layers, with the handling of local constraints typically being delegated to subroutines that extract annotation values from the corpus and compare them against those constraints.

4.2. Query Protocol

The architecture overview in Fig. 1 shows multiple (potentially very heterogeneous) service components that need to be able to efficiently communicate with each other during the query evaluation workflow. As such we decided to use JSON[10] as the basic transport format for our query protocol. It is a widely used and lightweight format, and its extension JSON-LD[11] also provides the means for strongly-typed transfer of complex data objects.
Queries in IQP are designed to be self-contained, i.e. they cover the entire information on **which** resource(s) to query, **how** to configure the evaluations engine, the actual query **constraints**, as well as instructions for preparing the **result** returned to the front-end. The following sections provide a brief introduction and examples for some of the main sections in any IQP query[12]. A mock-up query showcasing some of the protocol's features is shown in Fig. 2, parts of which are subsequently used to demonstrate the processing and partial evaluation of query payloads.

4.2.1. Preamble

Each query has a dedicated section that minimally defines the dialect of the query language to be used or defaults to the initial draft version. Beyond that, this preamble section can also contain several optional declarations: **Import** declarations extend the evaluation engine with additional features or modify existing behavior. Simple configuration of the evaluation workflow can be performed via **switches** and **properties**, for instance when disabling case-aware string matching or selecting the direction in which corpus elements should be traversed. Additionally, queries for IQP can embed **binary data** encoded in textual form to be used in query expressions, such as fragments of an audio stream.

[10]JavaScript Object Notation https://www.json.org

[11]JSON for Linked Data https://json-ld.org/

[12]A more comprehensive specification draft of the query language and the JSON-LD elements used in the protocol can be found online in the working repository (cf. Section 5).

```
1  {
2  "@type" : "iql:Payload",
3  "iql:queryType" : "singleLane",
4  "iql:lanes" : [ {
5   "@type" : "iql:Lane",
6   "iql:laneType" : "sequence",
7   "iql:elements" : [ {
8     "@type" : "iql:Node",
9     "iql:constraint" : {
10     "@type" : "iql:Predicate",
11     "iql:expression" : {
12      "@type" : "iql:Expression",
13      "iql:content" : "pos==stts.ADJ"
14     }
15    }
16   }, {
17   "@type" : "iql:Node",
18   "iql:constraint" : {
19    "@type" : "iql:Predicate",
20    "iql:expression" : {
21     "@type" : "iql:Expression",
22     "iql:content" : "[form==\"test\"]"
23    }
24   }
25  } ],
26  "iql:nodeArrangement" : "adjacent"
27 } ] }
```

Figure 3: Processed version of the payload expression shown in Fig. 2 in line 17. Most notably, the original query expression has been split into two separate node objects with embedded constraint expressions.

4.2.2. Streams

An IQL query contains at least one **stream** definition, typically to extract data from a single corpus. Multiple streams could be used to query for instance parallel corpora or multi-modal data that comprises different sets of primary data with some form of mapping between them. In the initial IQP implementation we will however restrict the engine to only evaluate single-stream queries and leave the extension to multiple streams for a later iteration.
Streams encompass the selection of layers from a corpus to be used, the binding of corpus members to usable variables in query expressions, result preparation instructions and the actual query constraints. Most of those components can be provided to IQP either fully preprocessed or in *raw* statements as described in the following section. Raw statements are automatically compiled during the preprocessing phase of the query evaluation, as described in Section 4.4. Inside a stream, constraints can be organized in so called lanes, where each lane provides access to a different (concurrent) structural or segmental layer.

4.2.3. Raw and Compiled Statements

When designing a query language and/or protocol, typically a compromise has to be made between succinctness, so that human users can easily write queries, and machine readability or completeness for the processing part. In IQL we support both sides equally:
The parts of a query that carry actual expressions for constraints, sorting or result instructions can be specified both in the form of *raw* statements or compiled objects. Fig. 2 shows an attribute `iql:rawPayload` in line 17 that contains the raw expression used to evaluate results. This is also the minimal form that a human user would have to type in a textual query interface. Subsequent preprocessing during the query evaluation turns this raw form into a more fine-grained separation of objects, visible in Fig. 3. Note how the entire expression has now been divided into nodes, constraints and expressions, that can be individually understood by the evaluation engine or back-end wrappers.

4.2.4. Solved Constraints

Wrappers for the different back-ends used for storage of a corpus are not expected to cover the full range of IQL expressiveness. As such the protocol needs a mechanism to mark already evaluated parts of a query on a very fine-grained level. Any constraint can be marked as `solved` and any element as `consumed`. Fig. 4 exemplifies this on the first node from Fig. 3 (lines 8 to 16). The constraint expression related to the first half of the word pair being an adjective has been marked as `solved` in line 6 with a value of `true` in line 7, meaning that all result candidates returned by the back-end wrapper are guaranteed to contain an adjective at the individually indicated word position. Subsequently, as all of its constraints are solved, the node itself is marked as `consumed` in line 3, allowing the engine to skip its repeated evaluation.
Assuming the back-end was not able to evaluate the second node (lines 17 to 24 in Fig. 3), this situation would now leave the IQP core to only test each candidate for having a word directly following the adjective with a surface form that matches “test” while ignoring case.

```
1  {
2  "@type" : "iql:Node",
3  "iql:consumed" : true,
4  "iql:constraint" : {
5    "@type" : "iql:Predicate",
6    "iql:solved" : true,
7    "iql:solvedAs" : true,
8    "iql:expression" : {
9      "@type" : "iql:Expression",
10     "iql:content" : "pos==stts.ADJ"
11 }}}
```

Figure 4: Example of a solved constraint as part of the answer send from a back-end wrapper to the IQP core.

4.3. Query Language

IQL build on concepts we previously described in Gärtner and Kuhn (2018) and uses a keyword-driven syntax to formulate complex query constraints in a way that is slightly verbose compared to other more compact CQL representatives. It does however provide greatly increased flexibility and basically an integrated scripting language to express constraints.
The two main features of IQL are structural constraints and constraint expressions. The latter can either occur within a structural constraint where they implicitly get access to

additional information and methods depending on the type of structure. Alternatively they can be used as global constraints in which case they are limited to bound corpus elements or globally available constants, methods and objects.

4.3.1. Constraint Expressions

Simply put, constraint expressions are arbitrarily complex expressions in IQL that evaluate to a Boolean result[13]. Since a complete introduction to the IQL grammar for expressions is not possible here, we only provide a few examples to highlight certain features. The expressions in Fig. 5 perform the following evaluations: (1) enclosing node is a noun, (2) lemma of enclosing node is one of the three listed movement-related verbs, (3) the bound node is a noun, (4) the parent node of a bound token has at least 5 children in total, (5) the part-of-speech tag of the second-to-last word in the bound sentence does not contain the symbol `N` (depending on the tagset, this will exclude nouns, proper nouns, conjunctions, past participle verbs and other tags at that position in the sentence).

```
1  pos == "NN"
2  lemma IN {"go","run","crawl"}
3  $token{"pos"} == "NN"
4  $token.parent.size() >= 5
5  $sentence.items[-2]{"pos"} !# "N"
```

Figure 5: Examples of constraint expressions in IQL.

4.3.2. Structural Constraints

Structural constraints define properties that target structures have to meet in order to be considered as result candidates. IQL supports three types of structural constraints, namely sequences, trees and graphs. Their occurrence within a query lane dictates the basic complexity and evaluation strategy for that part of the query, and they also cannot be mixed. In their basic form, structural constraints are always existentially quantified, but by using explicit quantifier statements they can also get existentially negated, universally quantified within their context or marked to occur a specific number of times in a match. Similarly to Section 4.3.1 the following list of examples is not exhaustive and merely intended as a brief overview on some of the structural constraint features available in IQL.
Fig. 6 shows three examples for each of the aforementioned types of structural constraints: (1) is a simple existentially quantified node definition, (2) explicitly quantifies a node to occur at least four times, (3) requires two to five nodes between `$x` and `$y`, (4) is a simple tree with existentially quantified nested child nodes, (5) existentially negates children in a tree node based on some constraint `x`, (6) universally quantifies a child constraint, meaning that all immediate child nodes must satisfy constraint `x`, (7) declares basic graph constraints via nodes and edges, (8) is a negated graph edge, and (9) finally shows the declaration of a explicitly quantified graph edge with its own local constraints.

[13]The specification also defines rules to optionally convert arbitrary primitive values or objects to Boolean values as well.

```
1  []
2  <4+>[]
3  [$x]<2-5>[][$y]
4  [[$x][$y[$z]]]
5  [ ![x]]
6  [ *[x]]
7  [x],[]---[y],[z]
8  []<--![x]
9  <4->[]--[x]->[]
```

Figure 6: Examples of structural constraints in IQL. Note that the angle brackets around numerical quantifiers are optional and only included for readability here.

4.3.3. Example Queries

In this section we demonstrate some of the expressive capabilities of IQL based on a series of information needs of varying complexity defined by Lai and Bird (2004) that have been used repeatedly in other work to compare CQLs and which are listed in Fig. 7.

Q1. Find sentences that include the word “saw”.
Q2. Find sentences that do not include the word “saw”.
Q3. Find noun phrases whose rightmost child is a noun.
Q4. Find verb phrases that contain a verb immediately followed by a noun phrase that is immediately followed by a prepositional phrase.
Q5. Find the first common ancestor of sequences of a noun phrase followed by a verb phrase.
Q6. Find a noun phrase which dominates a word “dark” that is dominated by an intermediate phrase that bears an L-tone.
Q7. Find an noun phrase dominated by a verb phrase. Return the subtree dominated by that noun phrase only.

Figure 7: Linguistic information needs for querying treebanks defined by Lai and Bird (2004).

The following example queries all assume that the sentence layer has been selected as the primary layer of the query scope and for reasons of simplicity we only show the inner query payloads for most of the examples.
Queries Q1 and Q2 can be expressed in sequence mode:
Q1. `FIND [form=="saw"]`
Q2. `FIND ![form=="saw"]`
From Q3 onward the tree mode can be used and structural constraints are expressed by nested node definitions:
Q3.
```
FIND [label=="NP"
     [last && label=="N"]]
```
The `last` keyword is an instruction that forces the engine to only consider the last item within the current context. Without this optimization the constraint could still be expressed by `endsWith(parent)` instead of the `last` keyword, but this would allow the engine to consider and then discard all but the last child during evaluation. Global constraints (used in Q5) provide another efficient way of defining this query by explicitly referencing the last child within the outer node and testing it for being a noun.

Q4. `FIND [label=="VP" ADJACENT [label=="V"][label=="NP"][label=="PP"]]`
With the `ADJACENT` arrangement for a set of nodes the engine will ensure that matches are adjacent[14] to each other in the order they have been declared in the query.
Q5.
```
FIND ADJACENT
     [$np label=="NP"][$vp label=="VP"]
   HAVING ancestor($np,$vp) AS $a
```
Tree matching in IQP is typically performed top-down, which is impractical in cases like this, where bottom-up evaluation is required to find the first node to match the ancestor constraint. Using the `HAVING`[15] keyword, global constraints can be defined which will be evaluated after the basic tree matcher has produced preliminary candidates. A collection of methods modeling tree relations is available to simplify queries such as this one[16]. The query can also be expressed using transitive dominance constraints and explicit (crosswise) negation, but it would (i) be very intricate and (ii) much less efficient to evaluate, as the engine again has to explore many false possibilities.
Q6.
```
WITH $w FROM tokens
    AND $np FROM syntax
    AND $ip FROM intonation FIND
  LANE syntax [$np label=="NP"
      [$w form=="dark"]]
  AND LANE intonation [$ip label=="L"
      && type=="IP" [$w]]
```
This example query contains the entire binding section to illustrate its usage. It also makes use of `LANE` declarations to access information from two different structural layers (here dubbed `syntax` and `intonation` for brevity) and joins them implicitly on the word level. Such a join could also be specified explicitly with global constraints similar to previous examples.
Q7. cannot be expressed fully in the initial IQL draft. As mentioned in Section 2 the specification of result processing instructions is planned for a later iteration. However, since IQL allows very fine-grained control over the referencing of individual parts in a match, restricting the result to only contain selected subparts will be a trivial matter.

4.4. Evaluation Engine

At the very core of IQP sits a custom evaluation engine that manages the preprocessing of queries, delegation to back-end wrappers if available, and most importantly the evaluation of any unsolved query constraints that remain and subsequent result preparation. Relevant parts of the query preprocessing and how (partially) solved constraints and consumed nodes are expressed in the protocol have already been mentioned in Section 4.2. In this section we will primarily and briefly present technical aspects of the evaluation engine that are related to performance and scalability. IQP builds on ICMF and as such uses the in-memory instances of its model during the evaluation process. For every query, a specialized automaton-like *matcher* is created that inspects each unit-of-interest (UoI) in the corpus independently and checks it for being a valid result candidate. In the case of a query focused on syntax, this would normally result in every sentence in the corpus being visited sequentially. Combined with the ability of ICMF to only load selected subparts of a corpus into memory, this enables a highly parallelizable query evaluation: Evaluation on a large corpus can be split over multiple computation nodes, each dealing with a selected region of the entire corpus resource. Within individual computation nodes (or if the engine only runs on a single machine), workload can also be efficiently split across available processor cores, as the evaluation of individual UoIs is independent of each other and so only minimal synchronization overhead is required.
Since this evaluation is performed index-less, the engine is essentially performing an uninformed brute-force search through the entire corpus, which (depending on the type of search and the complexity of query constraints) can potentially cause extremely long waiting times until the query result can be returned[17]. This issue can be offset to a certain degree with corpora being stored in database or text indexing systems with an associated back-end wrapper[18] that can at least handle a subset of IQL and thereby greatly reduce the number of UoIs the engine core has to inspect.

5. Availability

IQP is being developed as a set of Java libraries (requiring a Java 8 runtime environment) as part of the ICMF working repository. The code is freely available under an open source license on GitHub and a comprehensive specification of IQL is also part of the same repository. They all can be found in the general ICARUS2 repository group.[19]

6. Conclusion

In this paper we have presented a hybrid corpus query architecture to address the issue of continued fragmentation in the landscape of corpus query systems and languages. Taking the form of a middleware system between user front-ends and optional database or text indexing solutions as back-ends, it allows to decouple those two traditionally monolithically connected components of CQSs. With its novel corpus query protocol it guides a query evaluation workflow that allows partial solutions from back-ends to be taken into account in order to improve performance.

[14]Adjacency between arbitrary items in the ICARUS2 model is defined based on the mapping to their common foundation layer (if available), which typically contains the basic word tokens.

[15]Inspired by the SQL keyword with the same name, that also is used to extend the capabilities of a query beyond the filter operations of a `WHERE` clause (the equivalent of lanes and/or local constraints in IQL), using aggregate values.

[16]The labels $np, $vp and $a enable nodes to be referenced in additional expressions or global constraints and here are expected to have been bound to represent nodes in the constituency tree.

[17]Depending on sorting or other processing steps for the result, a just-in-time delivery of individual result chunks won't be feasible, as the engine might need the entire set of result candidates to be available first before deciding on which of them to actually return and in what order.

[18]If such a utility is not available for a large corpus, evaluation time could in fact be a prohibitive factor against the usage of IQP.

[19]The metadata behind this persistent identifier leads to both the repository and project pages: `http://hdl.handle.net/11022/1007-0000-0007-C636-D`

The current reference implementation is programmed in Java and strongly relies on ICMF for corpus interaction. The overall architecture, the query protocol and workflow however are not as strictly coupled to either of those two and as such the entire concept could also be transferred to other technology stacks.

Bibliographical References

Bański, P., Bingel, J., Diewald, N., Frick, E., Hanl, M., Kupietz, M., Pęzik, P., Schnober, C., and Witt, A. (2014). KorAP: The new corpus analysis platform at ids mannheim. Human language technology challenges for computer science and linguistics. 6th language & technology conference december 7-9, 2013, Poznań, Poland, pages 586 – 587, Poznań. Uniwersytet im. Adama Mickiewicza w Poznaniu.

Bański, P., Frick, E., and Witt, A. (2016). Corpus query lingua franca (CQLF). In Nicoletta Calzolari (Conference Chair), et al., editors, *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016)*, Paris, France, may. European Language Resources Association (ELRA).

Bingel, J. and Diewald, N., (2015). *KoralQuery – A General Corpus Query Protocol*, volume 111, pages 1–5. Linköping University Electronic Press.

Bodmer, F. (2005). Cosmas ii - recherchieren in den korpora des IDS. *Sprachreport : Informationen und Meinungen zur deutschen Sprache*, 21(3):2 – 5.

Christ, O. (1994). A modular and flexible architecture for an integrated corpus query system. In *Proceedings of COMPLEX'94: 3rd Conference on Computational Lexicography and Text Research*, pages 23–32, Budapest.

Clematide, S. (2015). Reflections and a proposal for a query and reporting language for richly annotated multiparallel corpora. In *Proceedings of the Workshop on Innovative Corpus Query and Visualization Tools at NODALIDA 2015, May 11-13, 2015, Vilnius, Lithuania*, number 111, pages 6–16. Linköping University Electronic Press, Linköpings universitet.

Diewald, N., Hanl, M., Margaretha, E., Bingel, J., Kupietz, M., Banski, P., and Witt, A., (2016). *KorAP Architecture – Diving in the Deep Sea of Corpus Data*, pages 3586–3591. European language resources distribution agency.

Evert, S. and Hardie, A. (2011). Twenty-first century Corpus Workbench: Updating a query architecture for the new millennium. In *Proceedings of the Corpus Linguistics 2011 conference*, Birmingham.

Gärtner, M. and Kuhn, J. (2018). A lightweight modeling middleware for corpus processing. In Nicoletta Calzolari (Conference chair), et al., editors, *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Paris, France, may. European Language Resources Association (ELRA).

Gärtner, M., Thiele, G., Seeker, W., Björkelund, A., and Kuhn, J. (2013). ICARUS – an extensible graphical search tool for dependency treebanks. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60, Sofia, Bulgaria, August. Association for Computational Linguistics.

Ghodke, S. and Bird, S. (2012). Fangorn: A system for querying very large treebanks. In *COLING 2012: Demonstration Papers*, pages 175–182, Mumbai, India, December.

Gärtner, M. and Kuhn, J. (2018). Making corpus querying ready for the future: Challenges and concepts. In *Proceedings of the 27th International Conference on Computational Linguistics*, KONVENS 2018, Wien, Österreich.

Kepser, S. (2003). Finite structure query: A tool for querying syntactically annotated corpora. In *Proceedings of the Tenth Conference on European Chapter of the Association for Computational Linguistics - Volume 1*, EACL '03, pages 179–186, Stroudsburg, PA, USA. Association for Computational Linguistics.

Kepser, S. (2004). Querying linguistic treebanks with monadic second-order logic in linear time. *Journal of Logic, Language and Information*, 13(4):457–470, Mar.

König, E. and Lezius, W. (2000). A description language for syntactically annotated corpora. In *Proceedings of the 18th Conference on Computational Linguistics - Volume 2*, COLING '00, pages 1056–1060, Stroudsburg, PA, USA. Association for Computational Linguistics.

Krause, T. and Zeldes, A. (2014). ANNIS3: A new architecture for generic corpus query and visualization. *Digital Scholarship in the Humanities*.

Lai, C. and Bird, S. (2004). Querying and updating treebanks: A critical survey and requirements analysis. In *In Proceedings of the Australasian Language Technology Workshop*, pages 139–146.

Lai, C. and Bird, S., (2005). *LPath+: A First-Order Complete Language for Linguistic Tree Query*. ACL Anthology, 12.

Lezius, W. (2002). *Ein Suchwerkzeug für syntaktisch annotierte Textkorpora*. Ph.D. thesis, IMS, University of Stuttgart. Arbeitspapiere des Instituts für Maschinelle Sprachverarbeitung (AIMS), volume 8, number 4.

Lundborg, J., Marek, T., and Volk, M. (2007). Using the Stockholm TreeAligner. In *6th Workshop on Treebanks and Linguistic Theories*.

Luotolahti, J., Kanerva, J., Pyysalo, S., and Ginter, F. (2015). SETS: Scalable and efficient tree search in dependency graphs. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*, pages 51–55, Denver, Colorado, June. Association for Computational Linguistics.

Marek, T., Lundborg, J., and Volk, M. (2008). Extending the TIGER query language with universal quantification. In *KONVENS 2008: 9. Konferenz zur Verarbeitung natürlicher Sprache*, pages 5–17, October.

Maryns, H. and Kepser, S. (2009). Monasearch – a tool for querying linguistic treebanks. In *Proceedings of TLT 2009*, Groningen.

Mueller, M. (2010). Towards a digital carrel: A report about corpus query tools.

Pajas, P. and Štěpánek, J. (2009). System for Querying

Syntactically Annotated Corpora. In *ACL-IJCNLP: Software Demonstrations*, pages 33–36, Suntec, Singapore.

Štěpánek, J. and Pajas, P. (2010). Querying diverse treebanks in a uniform way. In Nicoletta Calzolari (Conference Chair), et al., editors, *Proceedings of the Seventh conference on International Language Resources and Evaluation (LREC'10)*, Valletta, Malta, may. European Language Resources Association (ELRA).