Designing a Re-Usable and Embeddable Corpus Search Library

Thomas Krause[1], Ulf Leser[2], Anke Lüdeling[1], Stephan Druskat[1]
[1]Humboldt-Universität zu Berlin, Dept. of German Studies and Linguistics, Unter den Linden 6, D-10099 Berlin
[2]Humboldt-Universität zu Berlin, Dept. of Computer Science, Unter den Linden 6, D-10099 Berlin
krauseto@hu-berlin.de, leser@informatik.hu-berlin.de, {anke.luedeling, stephan.druskat}@hu-berlin.de

Abstract

This paper describes a fundamental re-design and extension of the existing general multi-layer corpus search tool ANNIS, which simplifies its re-use in other tools. This embeddable corpus search library is called graphANNIS and uses annotation graphs as its internal data model. It has a modular design, where each graph component can be implemented by a so-called graph storage and allows efficient reachability queries on each graph component. We show that using different implementations for different types of graphs is much more efficient than relying on a single strategy. Our approach unites the interoperable data model of a directed graph with adaptable and efficient implementations. We argue that graphANNIS can be a valuable building block for applications that need to embed some kind of search functionality on linguistically annotated corpora. Examples are annotation editors that need a search component to support agile corpus creation. The adaptability of graphANNIS, and its ability to support new kinds of annotation structures efficiently, could make such a re-use easier to achieve.

Keywords: corpus tools, multi-layer corpora, interoperability, graph database

1. Introduction

The creation of a search tool for linguistic corpora can be a large development effort. Nevertheless, new tools are still being developed, due to their necessity for studies in corpus linguistics as well as the increasing diversity of corpus types and phenomena available for study.[1] Exemplary reasons for such a re-development are:

(1) new kind of annotation structures not supported by any other tool,

(2) non-ergonomic integration of the existing tool,

(3) problematic software licenses, or

(4) performance problems with the existing tool.

Problem (1) holds true especially true for multi-layer corpora (Dipper, 2005), where different kinds of annotations are combined into the same corpus and where it can be expected that the corpus is extended with new types of annotation over time. Multi-layer corpus search tools like ANNIS (Krause and Zeldes, 2016) are often designed to support many kinds of annotation structures generically in the same software and query language. This generality in ANNIS is accomplished by using annotation graphs (Bird and Liberman, 2001) as the underlying data model. Despite the general data model, there are corpora which are difficult to represent in ANNIS, e.g., corpora for sign language, which need support for more complex concepts of tokens, temporal and spatial annotations (Hanke and Storz, 2008). Thus, even a generic multi-layer corpus search tool like ANNIS needs to be extended regularly to support more types of data. The problems (2) and (3) can occur if a non-search-centric tool, such as an annotation editor, needs to be extended with search functionality, e.g., because it is supposed to support an agile corpus creation workflow (Voormann and Gut, 2008; Druskat et al., 2017). Also, performance issues (4) often prove to be a permanent problem. While computer hardware evolves, the need for larger corpora with more tokens, more annotation layers and more relationships (even between documents if text reuse is studied (Berti et al., 2014)) increases as well, and keeping up with the amount of data poses a constant challenge.

When the need for a new corpus tool or integration of a query system in an existing software arises, it can be more sustainable to at least partially rely on an existing solution. Consider the following example: An annotation tool needs a custom user interface for presenting search results with tight integration to an existing editor. It would not necessarily need to implement a new query language or query engine, or a whole query system with all necessary components. Instead, it could simply re-use parts of existing domain-specific query systems. This paper describes a fundamental re-design and extension of the existing general multi-layer corpus search tool ANNIS, which simplifies its re-use in other tools.

2. graphANNIS

We want to present an approach to design an embeddable corpus search library that addresses the aforementioned problems, and discuss the benefits and downsides of this design. The library that will be used as a case study is the graphANNIS query engine, which is described in more detail in Krause et al. (2016). graphANNIS was used to re-implement the ANNIS Query Language (AQL) (Rosenfeld, 2010; Krause and Zeldes, 2016) as a main memory query engine in the C++ programming language. It can represent the same range of annotation types as the original ANNIS implementation, which includes

- token annotations and multiple tokenization,
- span annotations,

[1]Many dedicated and general search tools exists. Examples for generic search tools are CWB (Evert and Hardie, 2011), KorAP (Diewald and Margaretha, 2016), TIGERSearch (Lezius, 2002) or EXMARaLDA EXAKT (Schmidt and Wörner, 2014).

- dominance relations (e.g. for syntax trees), and
- pointing relations for generic edges between any annotation node.

While it supports AQL, the implementation is much more flexible compared to the original mapping of AQL to SQL. It uses directed graphs that are partitioned into acyclic components as its basic data model.
By using a main memory-based approach instead of a relational database, the question of how large corpora graphANNIS can support arises. In Krause et al. (2016), the used memory for different kind of corpora is reported. The "Parlamentsreden_Deutscher_Bundestag" corpus (Odebrecht, 2012) contains around 3.1 million tokens and each token has part-of-speech and lemma annotations. It uses less than 600 MB of main memory. Servers with 512 GB of main memory are available, and such a server could host corpora $\sim$ 850 times larger than the "Parlamentsreden_Deutscher_Bundestag" corpus. For such simple token-only corpora, the memory consumption of graphANNIS raises linearly with the number of tokens. Thus, even corpora with approximately 2.6 billion tokens, including part-of-speech and lemma annotation, should fit into the main memory of such a server. The size of a corpus cannot ultimately be defined by numbers of tokens alone. Instead, a depth factor must be taken into account, referring to the complexity of a corpus, that is, the numbers of annotation nodes and edges on top of tokens, and across different annotation layers. Deeply annotated corpora with a large number of layers and consequently a large number of nodes and/or relations can arguably be defined as "large".

3. Modular implementation

Each component of the annotation graph can be implemented in graphANNIS in a specialized module (see Figure 1 for an overview). Such a graph storage is optimized to find reachable nodes and distances between nodes inside a component. In contrast to the similar Ziggurat design (Evert and Hardie, 2015), the implementation optimization is agnostic to the type of annotation that is encoded in this graph and only depends on the graph structure itself. Query operators can be implemented by using the reachability and distance functions of these components to efficiently implement queries for linguistic annotation concepts like precedence or dominance. New operators can be added, which allows to add support for more query languages, or address currently missing features of AQL like the ones described in Frick et al. (2012).
GraphANNIS currently implements three different types of graph storages:

- one that stores the graph in an adjacency list and uses graph traversal for finding reachable nodes,
- a pre-/post-order based implementation based on the ideas of Grust et al. (2004), and
- a graph storage that stores linear graphs[2] by using a single order value per node.

[2]A Linear graph (or "path graph") is a tree where the maximum number of outgoing edges for a node is 1.

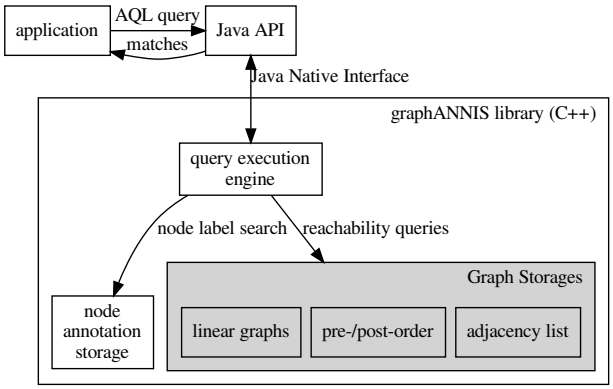

Figure 1: Overview of the graphANNIS library with its different graph storage modules.

The adjacency list is able to store all types of graph components, but the pre-/post-order based implementation is restricted to directed acyclic graphs and needs duplicate entries when the graph component is not a tree. When a corpus is imported, a heuristic is used to choose which graph storage best suits the given graph structure of each component. This modular design allows adding new types of graph storages when new annotations shall be supported by the query engine. These new graph storages can exploit the new types of graph structures and provide better performance than the existing implementations. For example, token precedence is modeled as explicit edges between tokens in graphANNIS. The length of the path between the first and the last token in the document is the number of tokens of the document. For queries that search for tokens that precede each other with indefinite length, a traversal on an adjacency list would be inefficient compared to direct lookup of an order value in a pre-/post-order encoding or the single order value of a linear graph. On the other hand, in cases where pre-/post-order encoding would result in duplicated entries because the annotation graph is not a tree, graph traversal can be more efficient instead.

4. Evaluation

In Krause et al. (2016), benchmarks have been executed to compare graphANNIS with the original relational database implementation of ANNIS. These benchmarks show that graphANNIS is around 40 times faster to execute a workload of more than 3,000 queries (collected from actual user interactions with the existing ANNIS system) from 17 corpora, than the relational database implementation. It could be argued, that a more monolithic graph-based implementation could handle the workload equally well. In order to test if our modular design and the specialized graph storages actually have a positive impact, additional benchmarks with a similar setup as in Krause et al. (2016), but on an updated set of queries, a have been performed.[3]

[3]We did not compare the performance with the relational database implementation in this paper because the focus is on the modularization. The data set including the queries will be released as part of a later publication, which will allow performing such a comparative benchmark.

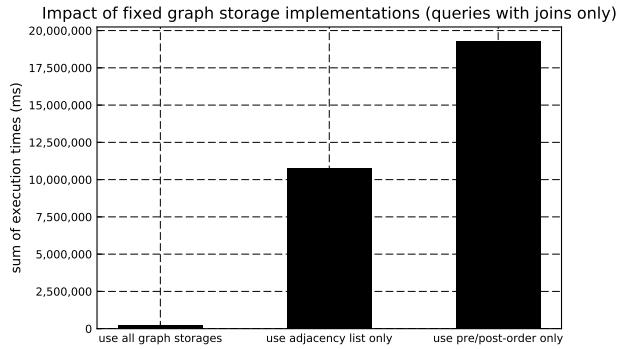

Figure 2: Impact of choosing different graph storage implementations per component on the execution time for the workload of 2,713 queries. Only queries that combine at least two annotation nodes haven been included in this workload. Overall performance is measured as the sum of the execution times for all queries in milliseconds. The baseline configuration (using all graph storages) executes the workload in $\approx 230,000$ ms.

The workload has been executed with the same version of graphANNIS on the same system, but with different optimization configurations. The configuration where all three graph storage implementations were enabled performed best (Figure 2). Another configuration, where the adjacency list graph storage has been used exclusively, was $>$40 times slower than the first one. Using graph indexes like the pre-/post-order alone does not help either, as this configuration is $>$80 times slower than the one with all graph storages enabled. There is no configuration where the graph storage for linear graphs has been used exclusively, because it is too specialized to represent all required graph types.
This experiment shows that the modularization actually has a positive effect. Our approach unites the interoperable data model of a directed graph with adaptable and efficient implementations. More detailed experiments to further investigate the strengths and potential problems of this approach are currently in progress.

5. Discussion and future work

While using a general data model is critical in supporting a wide range of possible annotation types, there are more practical issues that graphANNIS tries to solve as well. Reusing an existing query engine as software component can be challenging due to technical issues: If the system is only available as a web service for example, the developed software depends on the availability of this service, the network connection, and the sustainability of the infrastructure providing both. Even if the web service is open source, installing and managing it and all necessary dependencies (like a database management system) on a separate infrastructure can be overly complicated. graphANNIS is a software library that does not have dependencies that need to be installed separately. It is currently provided as both a C++ and a Java API. The library is also usable as OSGi[4] bundle, which made it possible to integrate it into the Atomic annotation tool (Druskat et al., 2014; Druskat et al., 2017).

Given the diverse landscape of corpus tools, providing only a C++ and a Java API does not seem sufficient. While a web service has the advantage of being indifferent to the programming language and operating system it is used by, an embedded software component can be more restricted in its ability to be integrated into these different types of systems. We therefore propose to extend graphANNIS with an API of the C programming language, which is supported by all major operating systems and programming languages. Such a C API would be a simplified interface to the functionality of graphANNIS and provide functions to execute queries (counting and finding instances), retrieving subgraphs, and administrative tasks like importing corpora. It would not expose the internal data structures like the actual graph storages.
As graphANNIS is available as an open source project licensed under the liberal Apache License, Version 2.0 on a public code hosting platform[5], the barriers for external contributors are already quite low. This measure also implies to foster sustainability and re-usability: when the integration into other programming languages or the feature set of graphANNIS is "almost sufficient" for an external project, it should be easier to extend graphANNIS than to develop a completely new system. GraphANNIS currently does not have some kind of dynamic plug-in system for extensions like adding new operators or graph storages, but changes by the community can be merged into the main code base. A possible problem for such a community-driven project is the usage of the C++ programming language, which is not widely used in other corpus linguistic projects. Extensive documentation, static code analysis, testing and continuous integration can help to make access easier and safer for new contributors. Alternatively, programming languages like Rust[6] provide more compile time guarantees for memory safety and absence of data races, with an execution efficiency similar to C++ and an easy way to provide an external C API. It should be evaluated if a port of graphANNIS to such a "safe" system programming language would be feasible and provide the same performance characteristics as the current C++ implementation. With the wide applicability of annotation graph data models for multi-layer corpora, efficient designs for graph search implementations on these models, techniques for easy integration into other tools and a community-driven development approach, the building blocks of a future interoperable linguistic query search library seem to be in sight and the development of such a system seems feasible.

6. Bibliographical References

Berti, M., Almas, B., Dubin, D., Franzini, G., Stoyanova, S., and Crane, G. R. (2014). The linked fragment: TEI and the encoding of text reuses of lost authors. *Journal of the Text Encoding Initiative*, (8).

Bird, S. and Liberman, M. (2001). A formal framework for linguistic annotation. *Speech Communication*, 33(1-2):23–60. Speech Annotation and Corpus Tools.

[4] https://www.osgi.org/

[5] https://github.com/thomaskrause/graphANNIS

[6] https://www.rust-lang.org

Diewald, N. and Margaretha, E. (2016). Krill: KorAP search and analysis engine. *JLCL*, 31(1):73–90.

Dipper, S. (2005). Xml-based stand-off representation and exploitation of multi-level linguistic annotation. In *Berliner XML Tage*, pages 39–50.

Druskat, S., Bierkandt, L., Gast, V., Rzymski, C., and Zipser, F. (2014). Atomic: An open-source software platform for multi-level corpus annotation. In *Proceedings of the 12th Konferenz zur Verarbeitung natürlicher Sprache (KONVENS 2014)*, volume 1, pages 228–234, Hildesheim.

Druskat, S., Krause, T., Odebrecht, C., and Zipser, F. (2017). Agile creation of multi-layer corpora with corpus-tools.org. In *DGfS-CL Poster Session. 39. Jahrestagung der Deutschen Gesellschaft für Sprachwissenschaft (DGfS)*, Saarbrücken, March.

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*. University of Birmingham.

Evert, S. and Hardie, A. (2015). Ziggurat: A new data model and indexing format for large annotated text corpora. *Challenges in the Management of Large Corpora (CMLC-3)*, page 21.

Frick, E., Schnober, C., and Banski, P. (2012). Evaluating query languages for a corpus processing system. In *LREC*, pages 2286–2294.

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

Hanke, T. and Storz, J. (2008). ilex–a database tool for integrating sign language corpus linguistics and sign language lexicography. In *LREC 2008 Workshop Proceedings. W 25: 3rd Workshop on the Representation and Processing of Sign Languages: Construction and Exploitation of Sign Language Corpora. Paris: ELRA*, pages 64–67.

Krause, T. and Zeldes, A. (2016). ANNIS3: A new architecture for generic corpus query and visualization. *Digital Scholarship in the Humanities*, 31(1):118–139.

Krause, T., Leser, U., and Lüdeling, A. (2016). graphANNIS: A Fast Query Engine for Deeply Annotated Linguistic Corpora. *JLCL*, 31(1):iii–25.

Lezius, W. (2002). TIGERSearch Ein Suchwerkzeug für Baumbanken. *Tagungsband zur Konvens*.

Odebrecht, C. (2012). Lexical Bundles. Eine korpuslinguistische Untersuchung. Master's thesis, Humboldt-Universität zu Berlin, Philosophische Fakultät II.

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

Schmidt, T. and Wörner, K. (2014). Exmaralda. In Ulrike Gut Jacques Durand et al., editors, *Handbook on Corpus Phonology*, pages 402–419. Oxford University Press.

Voormann, H. and Gut, U. (2008). Agile corpus creation. *Corpus Linguistics and Linguistic Theory*, 4(2):235–251.