Thomas Krause, Ulf Leser, Anke Lüdeling

graphANNIS: A Fast Query Engine for Deeply Annotated Linguistic Corpora

Abstract

We present graphANNIS, a fast implementation of the established query language AQL for dealing with deeply annotated linguistic corpora. AQL builds on a graph-based abstraction for modeling and exchanging linguistic data, yet all its current implementations use relational databases as storage layer. In contrast, graphANNIS directly implements the ANNIS graph data model in main memory. We show that the vast majority of the AQL functionality can be mapped to the basic operation of finding paths in a graph and present efficient implementations and index structures for this and all other required operations. We compare the performance of graphANNIS with that of the standard SQL-based implementation of AQL, using a workload of more than 3000 real-life queries on a set of 17 open corpora each with a size up to 3 Million tokens, whose annotations range from simple and linear part-of-speech tagging to deeply nested discourse structures. For the entire workload, graphANNIS is more than 40 times faster, and slower in less than 3% of the queries. graphANNIS as well as the workload and corpora used for evaluation are freely available at GitHub and the Zenodo Open Access archive.

1 Introduction

Many linguistic research questions require the analysis of several different types of information attached to different substructures of a given linguistic corpus. For instance, it is necessary to collate parts of speech and lemmas for the computation of complexity measures for language acquisition studies (Lue, 2011), to consider referential chains and intonation for the analysis of information structure phenomena (Baumann and Riester, 2013), and to combine temporal information and inflectional forms to trace language change (Hilpert, 2008). To make the categorization that lies behind such studies explicit, it is usually attached to a corpus as annotation of the different substructures, like tokens, phrases, or sentences (Leech, 1997; Lüdeling, 2011). A number of models and formats for representing and exchanging such annotated corpora have been proposed in recent years (Carletta et al., 2003; Chiarcos et al., 2008), with a clear tendency towards multi-layer models, i.e., models which allow the concurrent annotation of different types of (possibly nested) substructures. During creation and usage in research, such multi-layer corpora can grow by adding more texts and by adding more layers of annotation; as it is often not foreseeable at the beginning of a project what the

corpus may look like in a more advanced project state, systems for supporting the management and analysis of multi-layer corpora should ideally allow for extensibility both in the corpus model and in the corpus size.[1] Such a system should furthermore offer a powerful query language to perform quantitative analysis of corpora, and an implementation of this query language that is fast enough for interactive usage and that scales well with growing corpus size and complexity.

In this paper, we present graphANNIS, a very fast and flexible main-memory based implementation of the well-established corpus query language AQL (Rosenfeld, 2010; Krause and Zeldes, 2016). AQL was developed in the ANNIS project (Krause and Zeldes, 2016), whose long-term goal is to create a system that allows for unified search across diverse kinds of annotations. ANNIS supports a large range of different types and structures of annotations and offers an intuitive web-based user interface with sophisticated visualizations. The system is used as a corpus management platform for a variety of projects ranging from historic to spoken language. There are at least 11 public installations of ANNIS and an unknown number of non-public instances. Our own, publicly available ANNIS server (`https://korpling.org/annis3`) hosts more than 150 corpora, has 96 registered users and serves roughly 1800 queries every month.

The query language AQL builds on a graph-based data model, which makes modeling and querying deeply nested structures simple. However, the current implementation of AQL, called relANNIS, uses a relational database as storage and query layer. To this end, any AQL query is translated into one or more SQL commands which are executed by the database server. The tabular results are then transformed back into the AQL exchange format and displayed for user interaction at the ANNIS front-end. This design, which delegates the seamless exchange of data between main memory and disk to the database server, was originally chosen because memory was expensive at that time, which meant that very large corpora could not be maintained in main memory. The downside is it makes query execution somewhat slow, due to the necessary query and result transformation steps, the unsuitability of an SQL back end for certain graph-like predicates in AQL, and the general overhead incurred by the multi-tier, multi-user, transactional memory management inside a relational database[2]. Despite considerable effort spent on tuning and tweaking the system (Rosenfeld, 2010), including extensive use of indexing and materialized views leading to a very large disk footprint, we frequently observe queries that run into time-outs even on smaller corpora with less than $100,000$ tokens.

Here, we present graphANNIS, a novel implementation of AQL that directly implements its underlying graph-based data model and that is purely main-memory based. Due the ever falling prices of computer memory, main-memory technologies are nowadays extremely popular both in database research (Zhang et al., 2015) and applications (Färber et al., 2012). At the same time, especially deeply structured corpora are typically generated manually by linguistic experts, which limits their size to a few

[1] Example corpus projects where additional annotations have been added can be found e.g. in Zeldes (2016b) page 1f.

[2] Note that AQL is a pure query language and has no operations to manipulate data.

(dozen) megabyte. For instance, the size of the largest corpus managed by the public ANNIS server, the "Parlamentsreden_Deutscher_Bundestag" corpus (Odebrecht, 2011), is 500 megabyte in size (uncompressed) and has only flat annotations at token level. In contrast, the largest corpus in this collection which has a more complex annotation structure, the "TueBa-DZ 6.0" corpus (Telljohann et al., 2009b), has a size of only 1.2 gigabytes. Holding such corpora in main memory is easily possible with current PCs.

Graph-based databases are not a new topic, and several research (Wood, 2012) and commercial implementations (e.g., Neo4J, DEX) exist. However, systems are typically either optimized for navigational queries (adjacent nodes, neighborhood queries) but lack efficient support for transitive predicates (such as reachability queries), or are specialized data structures to support certain complex graph operations but lack basic navigational features. Therefore, we decided to build graphANNIS from scratch, which mainly encompasses a query compiler, a graph store, a basic query optimizer, and special index structures and graph algorithms to support specific features of AQL[3]. In this paper, we describe all of these components in detail and give a quantitative comparison of the performance of graphANNIS to that of relANNIS. We use a workload of more than 3300 AQL queries against 17 corpora; this workload was created by logging real-life user queries on the public ANNIS server over a period of roughly 5 months. Note that, to the best of our knowledge, this is the by far greatest real-life workload of linguistic queries ever published. We show that graphANNIS is more than 40 times faster than relANNIS over the entire workload; specifically, it outperforms relANNIS in 95% of all queries, and especially in deeply nested queries.

The paper is structured as follows: Chapter 2 describes the query language AQL and its existing implementation relANNIS. A description of the concepts underlying graphANNIS is given in Chapter 3. Chapter 4 describes the concrete implementation of graphANNIS. An evaluation how this new system compares to the legacy implementation relANNIS is presented in Chapter 5. Section 6 reviews other linguistic query systems, and Chapter 7 concludes our paper.

2 The ANNIS System for Corpus Querying

This section provides necessary background information that is needed to understand the design decisions that were made for the graphANNIS implementation. We describe the query language AQL and its current implementation relANNIS. For more details on both topics, the reader is referred to Rosenfeld (2010) and Krause and Zeldes (2016).

2.1 AQL, the ANNIS Query Language

AQL is a linguistically motivated query language not limited to a certain tag-set. It has been influenced by other query languages, such as TIGERSearch (Lezius, 2002),

[3] Note that graphANNIS is not yet a complete implementation of AQL, as it lacks certain convenience features; see Section 2.1 for details.

and is described more formally in Rosenfeld (2010) and Rosenfeld (2012). An up-to date description of AQL is also given in the ANNIS User Guide (Zeldes, 2016a).

The basic model of ANNIS is a graph consisting of nodes and edges. Nodes represent linguistic substructures of a text, like tokens, phrases, or sentences, whereas edges represent the relationship between nodes, such as followed-by, part-of, dominates, or within-distance. The type of a node is given by an annotation (or label).

The building blocks of an AQL query are terms, which select nodes either via a label or via their text. For instance, the query

```
pos="NN"
```

selects all nodes having the annotation (or label) "pos" (a common category name for part of speech annotations) with the value "NN" (often used for nouns). The query

```
"the"
```

will select tokens that have the value "the" as spanned text. In both types of selections, it is possible to use regular expressions instead of string literals by using `/` instead of quotation marks.

Labels can also have a namespace to distinguish annotations with the same name from different sources. This namespace can be selected using the `namespace:name` expression. For instance, the query

```
treetagger:pos="NN"
```

would select all noun tokens, where the POS information stems from the namespace "treetagger". This way, a single structure can have multiple values of the same type, such as multiple POS tags generated by different taggers or following different linguistic theories.

Multiple terms can be defined in one query using the special symbol `&`. Such conjunctions are typically followed by a predicate which specifies the nature of the selected nodes. For instance, the following query

```
cat="S" & "the" & #1 >* #2
```

first selects all nodes of category "S" (sentences) and then all nodes whose text is "the". Next, the Cartesian product of these two sets of nodes is built and filtered for those pairs where the sentence dominates (i.e., contains) the respective token. Note that the token need not be directly contained in the sentence; it might as well be that a sentence consists of phrases consisting of subphrases, etc. consisting of tokens. This implies that the path between the node with annotation "S" and the node "the" can have an arbitrary length. AQL also supports a number of more convenient syntax rules; for instance, the above query may also be abbreviated as

```
cat="S" >* "the"
```

Several other operators exist for different constraints, like paths lengths, types, and edge annotations (like direct dominance in syntax trees or co-reference chains). Another class of operators compare the texts covered by two nodes, such as phrase structures that overlap, or tokens being contained in a sentence. We describe the most important relationships in the following paragraphs.

Pointing relation operator: "`->type`" Each pointing relation has a named type and the names must be explicitly given as argument to the operator. All edges of one type define a graph component which is required to be acyclic. The path can be constrained to have the length 1 when using the "direct pointing relation" form of the operator (`->type`) or any length when using the indirect form which is marked by a star (`->type*`). An explicit range $n \dots m$ of allowed path lengths can be given by using the form `->type,n,m`. Only edges belonging to the same type are allowed in the path definition and there is no possibility to mix different types of edges by using only one operator. Edges are allowed to have labels as annotation and the direct pointing relation operator can take an additional argument that constrains the labels. An exemplary use of this predicate is the query

```
pos="VVFIN" ->dep[func="sbj"] tok
```

which selects all finite verbs (part of speech is "VVFIN") which are connected to a token via a pointing relation of type "dep" (which marks dependency relations). This connected token is the subject of the finite verb (this is marked with the `func="sbj"` edge annotation).

Dominance operator: "`>`" This operator selects a path composed of so-called dominance edges, which imply inclusion dependencies at the textual level. The syntax for the dominance operator is the same as for the pointing relation, thus it is possible to specify a path of unspecified length (`>*`), ranged length paths (`>n,m`) or edge labels (`>[anno="value"]`). An exemplary use of this predicate is the query

```
cat & cat="S" & #1 >[func="SB"] #2
```

which searches a sentence/clause category (node `#2` having a "cat" annotation with value "S") which is also a subject clause (expressed by the `func="SB"` edge annotation) and is dominated by another syntactic category (node `#1` having the "cat" annotation without a specific value).

Precedence operator: "`.`" Two tokens are in a precedence relationship if they are next to each other in the text. As with the other operators the precedence operator can have the path length as an argument. Precedence is not only defined for tokens but for any node that has a text coverage relation to any set of tokens. In this case the precedence is defined between the right-most covered token of the left-hand side (LHS) operand and the left-most covered token of the right-hand side (RHS) operand. For instance, in the sentence "I am hungry" the "I" token precedes the "am' token and the

"am" token precedes the "hungry" token. This construction in AQL would be expressed as

```
"I" . "am" . "hungry"
```

Text coverage operators: "_=_" "_o_" and "_i_" Any node covers a certain token span. Text coverage operators describe the relations between two text spans covered by a node. The identical coverage operator `_=_` ensures that both sets are completely equal while the overlap operator `_o_` only requires that there is a non-empty intersection between two sets. These operands may be swapped in a query since they describe a commutative relation between both nodes. In contrast, the inclusion operator `_i_` is not commutative since it defines that all covered tokens of the RHS must also be covered by the LHS.

graphANNIS implements all of the features of AQL described in this section. Additional features of AQL are available in the existing relANNIS implementation, including filtering documents by metadata, negation of values, filtering for value identity and more operators. For a complete list of features see the ANNIS User Guide (Zeldes, 2016a).

2.2 A Relational AQL Implementation: relANNIS

The existing relANNIS query system offers a complete implementation of AQL on top of the relational PostgreSQL (https://www.postgresql.org/) database. Queries entered via the ANNIS front-end are translated to SQL, executed by the database engine, and results are transformed from the relational data model to a Salt graph (Zipser et al., 2010) before being displayed through various visualizers in the front-end. Additionally, the total number of results of a query is computed by executing a separate SQL-query.

This workflow leaves most of the responsibility for optimizing the query execution to the relational database implementation. It also means that the graph model data in ANNIS needs to be mapped to the relational model. For most cases, this is straightforward; the database scheme of relANNIS uses mostly the following tables:

- `node` for informations about the node itself,
- `node_annotation` for the different labels of a node,
- `component` lists connected components,
- `rank` contains all pre-/post-order entries and
- `edge_annotation` for the labels of each edge.

Note that this is a normalized schema. It has some drawbacks; if an AQL query uses node annotations and edge labels, even a simple AQL operator between two annotations will cause an SQL-join with 9 tables (`edge_annotation` is joined once, all other tables are joined twice). However, also queries joining five annotations and more are common,

which would result in a query of more than a dozen joins. To avoid such difficult-to-optimize queries, relANNIS uses several materialized views that combine specific information into a single table. Using these tables, the query compiler can usually reduce the number of SQL-joins to match the number of AQL operators, but it also drastically increases the required space and makes memory management much harder for the database server.

A number of AQL queries are not as easily mapped to SQL. This affects, in particular, relationships between nodes over a path of unknown length, which cannot be expressed directly in SQL[4]. To answer such queries efficiently, relANNIS uses pre-/post-order indexing (Grust et al., 2004), with which reachability information can be obtained with a single query. However, this only holds true for tree-shaped graphs. To also handle annotations shaped as Directed Acyclic Graphs (DAGs), which are common in ANNIS, we use a technique described in Rosenfeld (2010), which increases memory consumption but still allows comparably fast queries.

An additional problem occurring in relANNIS are the statistics required by the cost-based optimizer in modern RDBMS. Note that each AQL operator results in at least one join, often over more than one attribute. The concrete way of executing such a join (and sizing of buffers etc.) is determined by the query engine based on statistics on the value distributions in the join columns. However, RDBMS typically assume these distributions to be independent (which allows for very succinct statistics and low cost maintenance), an assumption which is breached regularly in relANNIS; for instance, the post-order of a node will always be larger than the pre-order. We observed that, due to this discrepancy, PostgreSQL very often underestimates the size of intermediate results, which leads to suboptimal choices during query optimization.

3 Basic Design of graphANNIS

An important goal for the design of the new graphANNIS system was to reuse the best parts of the existing relANNIS system, but to address the performance issues which arose from using a disk-based relational database. The graph-based data-model and the query language AQL proved to be very useful for searching in linguistic corpora[5] and thus the new system should support the largest possible subset of AQL. Also the experience with the relational database showed that there should not be a mapping of data-models if not necessary. This helps to avoid losing possibilities for optimization, like good statistics for better execution plans. Therefore we did not want to rely on XML databases or RDF triple stores, as these have different data models.

Experiences with the query planning from PostgreSQL, where we needed to "trick" the optimizer into using a more efficient plan, also confirmed that we needed explicit control of the query execution. Such control is not possible when using a query language

[4] Note that the SQL-3 standard defines recursive queries which can express such operations. However, the current implementations are notoriously ineffective.

[5] For a discussion about the pros and cons of different linguistic query languages see Frick et al. (2012).

from a graph database like Cypher from Neo4j (Robinson et al., 2013). Also, systems like Neo4j usually have one specific way of storing and querying the graph, e.g. only using graph traversal. Since linguistically annotated graphs tend to have very different kinds of structures, and we want to be able to have solutions that are optimized for these specific structures, we did not want to restrict ourselves to a single storage and querying strategy.

ANNIS is solely a search system and using an SQL database to do the actual search helped to take advantage of all the advanced search optimization a modern relational database has to offer. But databases do a lot more than only providing search capabilities: They typically organize the persistent storage of the data on the disk and provide transactions for persistence and isolation of database updates. A corpus search system is special since after a corpus is imported once by an administrator, it solely provides read-only access to normal users. Thus complicated and costly transactions are not needed. The bulk import also allows to add much more computationally intensive optimizations since importing a corpus is not done very often. E.g., very specific index structures can be used and even data types for values can be optimized since the corpus is static and will not change. These index structures are the only way the read-only data will be accessed. Since there are no data updates, there is no need to store an additional normalized representation (e.g. the database table holding the nodes) if all information is available in the indexes themselves. We also wanted to avoid overhead created by managing the transparent persistence of the data to the disk.

General availability of notebooks with at least eight gigabytes of RAM and server systems with more than 100 gigabytes makes a main memory only approach feasible even for larger corpora. Not having to worry about the representation of the data both on the physical disk (which may not even be a relatively fast SSD) and the main memory allows to optimize the data structures for main memory and CPU cache access only. We also expect much faster queries if we can guarantee that all needed indexes are already loaded in main memory and do not need to be fetched from a disk. Our goal is to use data structures that, in contrast to the materialized view used in relANNIS (see section 2.2), scale well with the size of the corpus.

In order to achieve this goal we designed an architecture that combines the generic graph-based data model with specific implementations for different types of graphs. These so-called Graph Storages (GSs) only implement the storage of edge components while a common node annotation store stores all node-relevant information. Different annotation layers will result in different types of components and the architecture described here allows for the selection of an optimized implementation to store and query edges for a component. When executing a query it is parsed into a generic JSON representation, an optimized plan is generated with help of statistics and this optimized plan is subsequently executed. Finding nodes that are reachable from a starting point is an essential substep in these plans. The query optimizer does not need to choose which indexes to use, the best GS implementation is already chosen automatically when a corpus is first imported. Strategies for finding reachable nodes are graph-traversal, pre-/post-order or other graph indexes and our architecture allows to easily add new

Type	Description
COVERAGE	Edges between a span node and tokens. Implies text coverage.
INVERSE_COVERAGE	Edges between a token and a span node.
DOMINANCE	Edges between a structural node and any other structural node, span or token. Implies text coverage.
POINTING	Edge between any node.
ORDERING	Edge between two tokens implying that the source node comes before the target node in the text-flow.
LEFT_TOKEN	Explicit edge between any non-token node and the left-most token it covers.
RIGHT_TOKEN	Explicit edge between any non-token node and the right-most token it covers.

Table 1: Component types used in graphANNIS.

types of GSs.

4 Implementation

While the original ANNIS ecosystem is mostly developed in the Java programming language, graphANNIS is written in C++ 11. This allows for a better control of the allocation, deallocation and structure of the data in main memory. The system is optimized for read-only access of the data, thus the corpora are imported once and normally not changed after they have been loaded into the query system. Since the system is read-only there is no need for a complex transaction management. graphANNIS is implemented as a library which can be used in other programs and it is planned to make graphANNIS a drop-in replacement for the existing relational database back-end used in the current ANNIS versions.[6] All source code is published as open source at `https://github.com/thomaskrause/graphANNIS`.

4.1 Data model

graphANNIS represents the linguistic annotations as a directed labeled graph. The edges are grouped into named components and each component has an explicit type. A list of all component types used in graphANNIS is listed in Table 1. For some linguistic annotations like co-reference chains or constituent trees the mapping to the graphANNIS data model is straightforward: e.g., for a constituent tree the clauses are modeled as nodes, the tokens are terminal nodes and the relations are expressed using edges as can be seen in Figure 1. The data model of graphANNIS is very similar to the Salt (Zipser et al., 2010) data model, differing slightly only to match the semantics of AQL as closely as possible. E.g., both Salt and graphANNIS use explicit edges between

[6] In order to use the C++ library from the existing Java-based back-end the JavaCPP library (`https://github.com/bytedeco/javacpp`) is used.

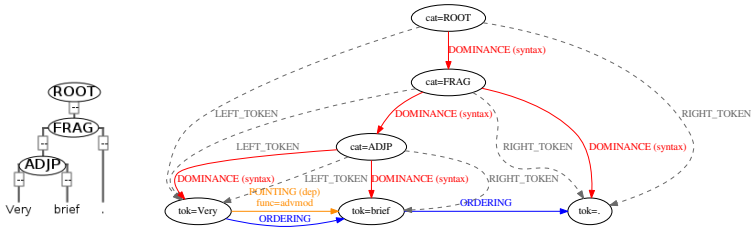

Figure 1: Example how a constituent tree is modeled in graphANNIS. Each token has the special annotation "tok" which contains the covered text as value and tokens are connected with explicit edges which define their ordering (in blue). So-called structural nodes of the constituent tree have their annotation as label and these are connected with edges of type DOMINANCE (in red) to either other structural nodes or tokens. In addition each non-token node has an explicit LEFT_TOKEN and RIGHT_TOKEN (the dashed lines) to the left- or right-most token they cover. There is also an additional dependency edge in this example which is modeled as a so-called pointing relation (in orange). This example is taken from the GUM corpus (Zeldes, 2016b) and is also available online in ANNIS (`https://korpling.org/annis3/?id=cc06121a-09b6-455f-aef4-f9eaee7a34f5`)

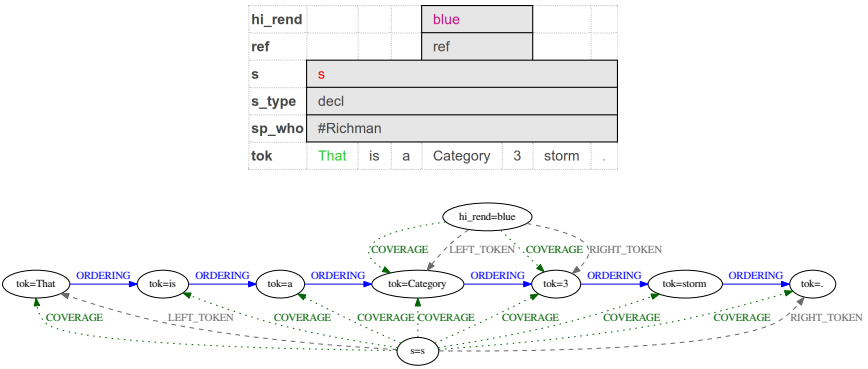

Figure 2: Example how spans are modeled in graphANNIS. Each span has an explicit edge to each token it covers (green COVERAGE edges). Only the "s" and "hi_rend" annotations have been included and the INVERSE_COVERAGE edges have been excluded to enhance readability. This example is taken from the GUM corpus and is also available online in ANNIS (`https://korpling.org/annis3/?id=d23627b4-3b30-4876-9ce5-7acb3406d3a4`)

the so-called span nodes (nodes that cover a range of tokens) from each span node to every token and this is compatible with the way AQL handles text coverage (see Figure 2 for an example). In AQL the leaf nodes in the annotation graph are always tokens whereas in Salt there are special nodes containing the complete text of a document and are connected with the tokens. The tokens in graphANNIS have a special label with the name "annis4_internal:tok" instead which contains the covered text.

4.2 Node labels

The graphANNIS implementation separates the storage of nodes and edges. There is one central map for storing node labels and several storages for different types of edges (see subsection 4.3 for details). Nodes are not explicitly stored but have a numeric ID to which node labels and edges refer. Every node has at least the special "annis4_internal:name" label, so it is always possible to iterate over all nodes even if they do not have a user-supplied annotation.

The query system should support corpora of all kinds of languages and thus needs to support Unicode in order to be able to represent the different scripts. In addition, strings representing linguistic annotation names or values might have a wide variation in their length. In certain tag sets the annotation names are very short like "pos" for "part of speech" and the possible annotation values are also abbreviations with a limited length. A good example for this kind of annotations is the Stuttgart-Tübingen-Tagset (STTS) (Schiller et al., 1999). As another extreme annotations can be free text comments with several hundreds of characters (e.g., the "Open-ended comments" in the CityU corpus from Lee et al. (2015)). graphANNIS uses a dictionary encoding where all strings get a unique ID and are only referenced by this ID. A global ID manager is responsible for adding new strings and getting the ID for a certain string or the string value of an ID. This has the advantage that an annotation has a constant size independently of the length of the strings it contains. For annotation categories with a limited number of possible values this approach might also reduce the needed memory, since repeated occurrences of a value or name only store the ID and not the real string. This only holds for annotation values or names whose memory representation is larger than the size of an ID (which is 4 bytes). The assumption does not hold for certain tag sets which contain very short identifiers representable with the ASCII encoding and will actually cause a slightly worse memory usage in this case. E.g., in the TIGER corpus (Brants et al., 2004) the phrase category annotations have values from 1 to 4, but typically only 2 characters.

Additionally to the actual values, the node label storage contains statistics about the values of different label categories. A label category is a combination of a namespace and name. These statistics include the total number of labels and equi-width histograms of the values for each label category. Thus it is possible to estimate the instance count of label categories for a specific value or a range of possible values both efficiently and accurately based on the actual distribution of label values in the corpus.

4.3 Specialized Graph Storages

The main task of the query system is to find nodes with certain labels and to check constraints on the path. Thus the node described by a RHS of the operator must be reachable by the node of the LHS. Finding nodes that are (indirectly) reachable by another node is therefore a very important subtask that must be implemented efficiently. There exist various graph indexes to query reachability (Grust et al., 2004; Seufert et al., 2013; Yildirim et al., 2010) but they typically do not work on any kind of graph (e.g., see section 2.2 for a description of the problems that pre-/post-order indexes have when used for a DAG instead of a tree). Since the annotation edges of different kinds of linguistic annotations are grouped in components, graphANNIS can exploit the specific structure of each component and use an optimized implementation for storing and querying edges.

A so-called Graph Storage (GS) is responsible to store the information about the edges of a component. It does not only allow to retrieve the directly connected edges for a node and their annotations, but provides more complicated functions for accessing the graph. Each GS has the functions (pseudo-code)

- `boolean isConnected(n1, n2, minDistance, maxDistance)`,
- `iterator findConnected(n1, minDistance, maxDistance)` and
- `integer distance(n1, n2)`.

`isConnected(...)` checks if node $n2$ is reachable from node $n1$ within a given (optional) distance. `findConnected(...)` will find all reachable nodes from a start node and `distance(...)` calculates the minimal path length between two given nodes. It is expected for each GS to have an optimized implementation for these three functions. As a result a GS cannot store any type of component but is limited to graphs which fulfill special constraints. It is the responsibility of the general database to only use an appropriate implementation for a specific component. There are three different kinds of graph storages implemented already.

4.3.1 Adjacency list

This implementation of a GS stores the edges as set of node ID pairs. The set implementation is based on B-trees and uses the Google C++ B-tree template classes.[7] It is guaranteed that the set is sorted and since the pair always starts with the source node it is possible to get all outgoing edges for a certain node in logarithmic time.

The three basic GS functions are implemented using a Depth-First-Search (DFS) graph traversal. If the function `isConnected(...)` is called for the exact distance of 1 a shortcut is used which only checks if the source/target node pair is contained in the set. Since the components in this implementations are allowed to contain cycles a cycle-safe implementation of DFS can detect cycles and will result in an error state if such a cycle is found. Using a cycle check makes the execution less efficient. The

[7] The library is available from `https://code.google.com/archive/p/cpp-btree/`

statistics of a component already contain the information whether a component is cyclic (see section 4.3.4) and this could be used to build a specialized variant of the adjacency list that can be used for read-only noncyclic components.

Since adjacency lists and cycle-safe DFS work for every graph regardless of its properties this implementation is used as a fall-back if no optimized implementation is available.

4.3.2 Pre- and post-order

As described in Section 2.2 the legacy ANNIS implementation is based on a relational database and uses a pre-/post-order based index to find reachable nodes efficiently. This GS implementation uses the same index to accelerate reachability queries. It does not support components that contain cycles. In order to store the hierarchical structure of the graph it maintains a map from a node ID to a list of triples containing the level and the pre- and post-order. There is also an inverse map for each distinct triple to the corresponding node ID. Both maps are based on B-trees.

4.3.3 Linear graphs

Linear graphs are graphs where each node can only have one ingoing and one outgoing edge at maximum. This is quite common in linguistic annotations, e.g. when the order of words is stored or in any other form of chains of references. When determining whether a node is a descendant of another node an ordering can be used. A component might have several root nodes and the order is always the distance of a node to its root node. The order of each node is stored in the map which maps the node ID to a pair containing the ID of the root node and the (relative) order of the node. Additionally there is a vector for each root node. Each entry in this vector consists of the ID of a connected node and the index inside the vector is the order of this node.

4.3.4 Automatic selection of best Graph Storage

Each component has different characteristics depending on which annotation scheme was used to create it. graphANNIS tries to exploit this and provide optimized implementations for certain typical structures that can be found in linguistic annotations. When an existing corpus is imported for the first time each component is stored in a fall-back implementation using adjacency lists. This implementation is able to calculate statistics on the component which helps to automatically decide which implementation can be used. Table 2 contains the fields of the collected statistics. After the import is finished each component is converted to an optimized implementation if possible.

The GS for linear graphs (see section 4.3.3) is chosen whenever the component is a forest of rooted trees (`rootedTree` is true) and the maximal fan-out (`maxFanOut`) is 1. Since we know the length of the longest path inside the component (`maxDepth`) and the order of a node inside a linear graph can be only as large as the longest path, we choose a memory-saving data type for representing the order. Depending on `maxDepth` a single

Field	Description
cyclic	True if the component contains cycles.
rootedTree	True if the component is a forest of rooted trees (tree having only one root node). This is checked by ensuring that the graph is not cyclic and that each node has maximally one incoming edge.
nodes	Number of nodes in this component.
avgFanOut	The average number of outgoing edges for each node.
maxFanOut	The maximal number of outgoing edges for each node.
maxDepth	If noncyclic, the length of the longest path.
dfsVisitRatio	If noncyclic, the ratio between the number of nodes and the number of visits when traversing each subcomponent in a Depth-First-Search.

Table 2: Information collected as statistics for each component.

order value is using either 8, 16 or 32 bytes of main memory. Since the linear graph GS is read-only there is no possibility that the maximal path size changes and does not fit any longer in the chosen data type.

When the component is a forest of rooted trees but the maximal fan-out is greater than 1, the pre- and post-order based GS is chosen. As with the linear graphs, a memory optimized implementation is chosen depending on the actual number of nodes and the maximal depth. The data type for the pre/post-order is limited by the number of nodes since every node can have only one pre/post-order in a rooted tree. In the case that the component is not a forest of rooted trees but still acyclic, it is checked whether the ratio between the number of nodes and the number of visits in a Depth-First-Search is greater than a constant (currently 1.03). This condition shall express the case that a component is "almost" a tree, thus the number of order entries for the nodes is expected to be low. If a component fulfills this condition the pre- and post-order GS is still used, but only the data type for the level of a node is optimized with the help of the `maxDepth` statistic.

The default adjacency list implementation is not only used as a fall-back but also in the case if the longest path of the component is 1. This is justified by the overhead other implementations have for storing and querying the graph. If the longest path is 1, the adjacency list can implement the three basic functions with a single map look-up and does not have the overhead of the other implementations.

4.4 Query Optimization

graphANNIS does not include its own parser for AQL queries. Instead, the original Java-based parser from relANNIS is used to create a intermediate representation of the query as JSON objects, which is then passed as argument to the graphANNIS library to construct and optimize an execution plan. An execution plan of an AQL query is a tree of operators like joins, path constraints, filters, or matching label values. Nodes with certain values in their labels can be searched using exact matches or regular expressions, and any search may make use of namespace information.

For each plan, graphANNIS tries to derive an abstract cost measure which correlates to the sum of processed tuples over all execution steps. Estimates are calculated by using statistics on the distribution of label values and edge types; the edge component statistics are the ones described in Table 2 and are used to estimate the outcomes of join and filter operations. For estimating the number of labels which fulfill a certain criterion histogram statistics are used (see section 4.2).

These cost estimates are used by a simple query optimizer to reorganize the original plan following the simple heuristic that plans creating smaller intermediate results are generally faster. In a first optimization step the operands of each commutative operator are switched if the estimated number of matches for the LHS is larger than the RHS, since a smaller LHS is beneficial for most joins. The second step is to optimize the order of the joins themselves. For queries with less than eight predicates we exhaustively enumerate all permutations to find the presumably best plan; for queries with more than 7 operators we use a simple genetic optimization algorithm to keep the search space at a reasonable size.

5 Evaluation

In this section, we compare the performance achieved by graphANNIS with that of relANNIS using a large real-life workload and a diverse set of different corpora. We also show that the main memory requirements for graphANNIS are acceptable for all multi-layer corpora assuming availability of a well-equipped desktop computer or of a smaller server.

5.1 Dataset used for benchmark

To test AQL implementations with a large, diverse and realistic set of queries, we anonymously collected AQL queries from our public ANNIS server. Note that this server provides both selected free corpora and access-restricted corpora (via a login system); for our tests, we only used queries against corpora which, in principle, are freely available[8]. The selection of corpora available on this server is based on actual research questions and we assume the queries executed on the system are relevant for the researchers using the server.

For each query only the query itself, the selected corpora, a time-stamp and the execution time were logged (but not the user). We collected data beginning in November 2015 and ended the collection in March 2016. In total 8584 unique queries were collected[9]; some data unfortunately was lost due to configuration problems. We filtered all queries which (1) targeted more than one corpus (AQL allows multi-corpus queries, (2) targeted a corpus which was used only rarely, or (3) used an AQL feature not yet available

[8] The licensing situation for corpora is not always clear. Some corpus creators disallow non-academic use and allow download of the original corpus files only after registration. Accordingly, not all corpora in our benchmark set are directly downloadable, but all are free for non-commercial use.

[9] By "unique" we mean syntactic equivalence.

	queries
BeMaTaC_L1_2013-02.1 (Sauer, 2013)	194
BeMaTaC_L2_2013-02.1 (Sauer, 2013)	87
DDD-Tatian (Donhauser et al., 2015)	201
falkoEssayL1v2.3 (Reznicek et al., 2012)	124
falkoEssayL2v2.4 (Reznicek et al., 2012)	360
FalkoWHIGL2v2.1 (Hirschmann et al., 2008)	28
Fuerstinnenkorrespondenz1.1 (Lühr et al., 2015)	131
HIPKON (Coniglio et al., 2014)	28
KAJUK (Ágel and Hennig, 2014)	31
kobaltL1v1.4 (Zinsmeister et al., 2012)	173
kobaltL2v1.4 (Zinsmeister et al., 2012)	360
Maerchenkorpus (Walter, 2015)	111
Parlamentsreden_Deutscher_Bundestag (Odebrecht, 2011)	734
pcc176 (Stede and Neumann, 2014)	465
RIDGES_Herbology_Version4.1 (Odebrecht et al., 2016)	244
tiger2 (Brants et al., 2004)	14
TueBa-DZ.6.0 (Telljohann et al., 2009a)	102
sum	3387

Table 3: Corpora used in the benchmark and the number of queries for each corpus.

in graphANNIS. Table 3 lists the names of the remaining corpora and the number of unique queries for each corpus. In total 3387 queries from 17 corpora were included in the benchmark.[10] Note that the workload contains each query only once; multiple executions are not measured, although they exist in the log files.

5.2 Benchmark setup

All runtime measurements were executed on a server with 16GB of DDR3 RAM (1333 MHz) and an Intel Xeon X3460 CPU clocked at 2.80GHz on an Ubuntu Linux system. Table 4 contains the versions of the used software.[11] PostgreSQL was configured to use a shared buffer with 8GB which is enough to hold the relevant tables and indexes of each corpus in main memory.

relANNIS includes an internal benchmark functionality where all queries for a single corpus are executed first in sequential order to make sure the data is in the database cache, and then executed again 5 times in random order. We used this feature and report the median of the last five executions as runtime of a query. The median was chosen to flatten out any outliers caused by tuples not yet included in the cache. Note

[10] From the original 8584 queries 25.7% were filtered out because the corpus was not included, 31,1% of the remaining queries did target more than one corpus and 22,8% of them were filtered out because they did use a not yet available AQL feature. The complete dataset including the queries and the benchmark results can be downloaded from `http://dx.doi.org/10.5281/zenodo.61807`. All open access corpora used in the benchmark can be downloaded from `http://dx.doi.org/10.5281/zenodo.154343`.

[11] The version of graphANNIS used for the benchmarks can be downloaded from `http://dx.doi.org/10.5281/zenodo.61811`.

Software component	Version
relANNIS	3.4.1
PostgreSQL	9.4.6
graphANNIS	benchmark-journal-2016-07-27

Table 4: Versions of the software components used in the benchmark.

	sum (in ms)
baseline	6637917
graphANNIS	161160

Table 5: Sum of workload execution times. This is the sum over the execution times of all the queries in the benchmark.

that this is a fairly RDBMS-friendly setup, as running all queries first allows filling the main memory caches with all relevant tuples. Queries were aborted after a 60 seconds time-out and aborted queries were counted as 60 seconds execution time.

graphANNIS also has an integrated benchmark mode using the Celero Benchmark library (`https://github.com/DigitalInBlue/Celero`). Each query in the dataset was converted to the JSON intermediate representation together with the execution time baseline from relANNIS. Then all queries were executed 5 times with the mean execution time as the result. Additionally the memory consumption for each corpus was measured. In contrast to relANNIS there is no warm-up phase necessary for graphANNIS (the complete data is already loaded in memory before the benchmark is started) and thus the mean execution time was chosen instead of the median.

5.3 Benchmark results

In Table 5 the sum of the execution times for the relANNIS baseline and graphANNIS are shown. For executing the entire benchmark, relANNIS requires 41.188 times more time than graphANNIS. The individual speed-ups are shown in Figure 3: 94.12% of the queries are faster in graphANNIS than when using relANNIS. Table 6 shows the quantiles of the speed-ups. For instance, 75% of the queries are at least 29.57 times faster in graphANNIS than in relANNIS.

While these figures clearly show the superiority of graphANNIS in terms of execution speed, one also has to consider main memory requirements, as graphANNIS must hold an entire corpus in main memory before being able to execute AQL queries. Table 7 lists the number of nodes and the memory usage of graphANNIS for each corpus in the benchmark set. Note that the used memory size does not only depend on the number of tokens but on the overall number of nodes and edges, which grow superlinearly in multi-layer corpora. Thus a corpus with a relatively small number of tokens like "falkoEssayL2v2.4" (144.619 tokens) can have a similar size to a corpus like "tiger2" (888.578 tokens) due to the fact that the annotation in "tiger2" is only based on tokens and a single syntax layer, while the Falko corpus contains a lot of spanning annotations.

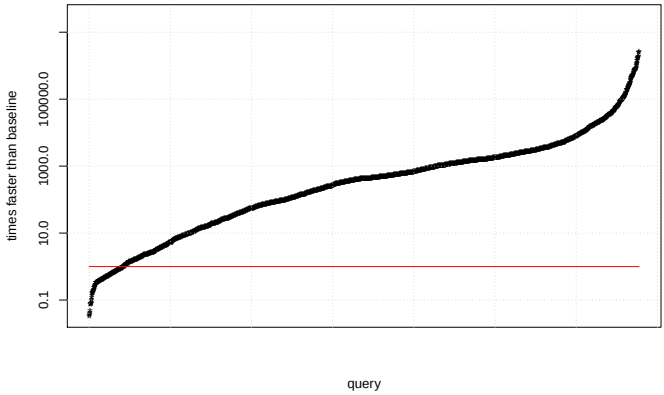

Figure 3: Distribution of the query execution time when using graphANNIS in comparison to the baseline relational database implementation. Each data point corresponds to a single query and how much time it took in relation to the baseline. Thus every data point below the red line (at 1.0) is slower than the baseline and the ones above are faster.

percent of queries	times faster than baseline
0%	2911623.49
25%	2097.62
50%	460.23
75%	29.57
100%	0.03

Table 6: Quantiles of the speedup from using graphANNIS instead of the baseline relational database implementation (larger is better).

Overall, even the largest corpus in this set, the TuebaD/Z corpus which contains 975, 836 tokens in version 6 leading to 15, 773, 656 nodes[12], requires only 1.6GB of memory.

There are still types of queries where graphANNIS is slower compared to the relational database implementation. E.g. the query which performs 28.59 times worse compared to the baseline (and thus is the worst query in the benchmark set) is `node & node & #1 ->dep[func="ART"] #2` : two very non-selective attribute definitions and an operator with a very selective edge annotation in between. Currently graphANNIS has to check all nodes if they have an outgoing edge that matches this edge annotation. This kind of query can be made much faster by adding an index which maps each edge annotation to its corresponding edge (with its source and target

[12] This version of TuebaD/Z contains several annotation layers like part of speech, constituent trees, and co-reference chains.

corpus	used memory (MB)	# node labels	# token
BeMaTaC_L1_2013-02.1	17.86	235563	11187
BeMaTaC_L2_2013-02.1	23.12	257490	12517
DDD-Tatian	43.96	849796	54677
falkoEssayL1v2.3	194.18	3639621	70615
falkoEssayL2v2.4	417.71	8103560	144619
FalkoWHIGL2v2.1	258.52	5593364	130949
Fuerstinnenkorrespondenz1.1	286.87	5043000	262465
HIPKON	40.15	727485	109045
KAJUK	65.66	907774	119420
kobaltL1v1.4	67.31	1186691	12984
kobaltL2v1.4	165.59	3033369	33368
Maerchenkorpus	55.90	1479400	295880
Parlamentsreden_Deutscher_Bundestag	588.53	15670960	3134192
pcc176	25.51	321544	33298
RIDGES_Herbology_Version4.1	223.94	3867351	154267
tiger2	427.01	6451776	888578
TueBa-DZ.6.0	1615.36	15773656	975836

Table 7: Main memory usage of the corpora in megabytes. The number of node labels and token is given as reference.

node ID) and using this index as an iterator for the LHS of the join. From the 25 slowest queries (compared to baseline) 19 queries match this pattern. Another typical problem are queries containing regular expressions that PostgreSQL can replace with index-optimized `LIKE` queries. For simple cases this is already supported in graphANNIS as well, but more complex regular expressions like `(N.|ART)` can result in a full search for all nodes with the annotation name even if a prefix based index lookup could be used. Figure 4 indicates that there is also some correlation between the number of attributes used in a query and the execution time. The number of attributes can be seen as an indicator for the complexity of a query since more attributes result in more joins. When the query statistics work well and the join order is optimized to produce as small intermediate results as possible the negative effect of the joins can be compensated. Additionally, queries with a larger number of attributes are less frequent in the workload. graphANNIS is able to handle the different query complexities and is faster than the baseline implementation even if queries having the same number of attributes are grouped into separate workloads (see Figure 5). Still there is room for improvement in join performance, as queries containing no join perform much better than the ones containing at least one join.

6 Related Work

There already exist several linguistic query systems which can be categorized via their data model and the expressiveness of their query language. The IMS Open Corpus Workbench (CWB) (Evert and Hardie, 2011) for example allows searching on annotations on tokens (positional annotations) and ranges of tokens (structural

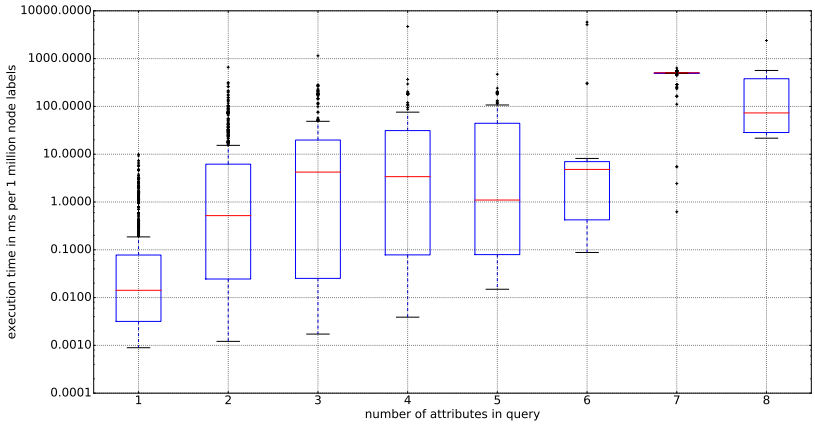

Figure 4: Box plot of the distribution of normalized execution times for each query grouped by the number of attributes a query has. To make the results comparable between different corpora, they have been normalized to the corpus size. The execution times are measured in milliseconds per 1 million node labels of the corresponding corpus.

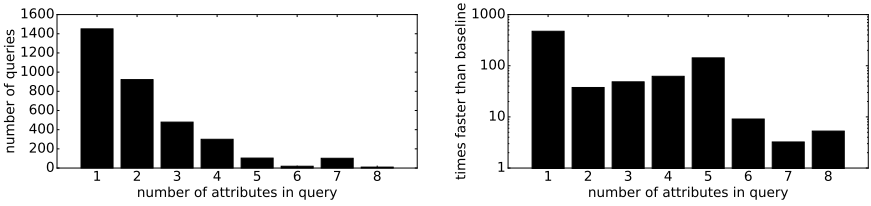

Figure 5: Distribution of number of attributes per query in the workload (on the left side) and speedup of graphANNIS compared to the baseline when the workload is grouped by the number of attributes per query (right side).

annotations). This is a rather "flat" annotation scheme which cannot properly encode general trees.[13] CWBs strength is the support for very large corpora: the authors of CWB claim they can support up to 2.1 billion words (see Evert and Hardie (2011) page 12f.). In contrast TIGERSearch (Lezius, 2002) was designed to handle a collection of sentences which are annotated with syntax trees according to the TIGER annotation scheme (Brants et al., 2004). TIGERSearch uses an index on the label values to restrict the search to single sentences, but it does not use an index on the tree structure itself. Since annotations are always limited to a single sentence TIGERSearch can still perform a very fast traversal search on each sentence and take advantage of the reduced search space. The approach to only use a single sentence as the search space is shared by other so called TreeBank systems like Tregex (Levy and Andrew, 2006) or tgrep2 (Rohde, 2005). The TüNDRA system (Losemann and Martens, 2012) uses the same query language as TIGERSearch and supports the same data model and input files. Queries are translated from TIGERSearch into XQuery and TüNDRA uses the XML database BaseX to perform the actual search. The KorAP system (Bański et al., 2014) also uses an external library to search the data. It can either use a Lucene based implementation (called Krill) or a Neo4j graph database. In both cases queries can be formulated in different query languages and are transformed into a unified representation which is executed by one of the back-ends. One of the query languages that is currently supported by KorAP is AQL.

In contrast to e.g. KorAP or CWB, graphANNIS is a main memory based system and thus the corpus size it can handle is limited by the amount of main memory available on the computer. [14] Another approach for a query system based on main memory was the ANNIS implementation from Rosenfeld (2012) on top of the MonetDB (`https://www.monetdb.org/`) column store. In this approach the original normalized relational database schema of ANNIS was used but SQL-queries were optimized to allow efficient joins in MonetDB. Since MonetDB also does not use any secondary indexes but sorts the tables by a column, overhead in representing the data was avoided. In the experiments it could be shown that using MonetDB allowed to execute the workload of selected queries on the Tiger2 corpus three times faster than PostgreSQL. At the same time the memory consumption for representing the Tiger2 corpus was about 400MB using MonetDB instead of the 8GB needed to store all tables and indexes in PostgreSQL.

7 Conclusions

This work presented graphANNIS, a main memory based search system for deeply annotated linguistic corpora implementing the popular AQL query language. We argued

[13] Evert and Hardie (2015) describe a not yet released new generation of the Corpus Workbench, where hierarchical and graph structures are supported.

[14] Estimating the maximal corpus size supported by graphANNIS is hard, since this not only depends on the number of tokens but also on the annotation layers of the corpus. E.g. for a Tiger2 like corpus graphANNIS could handle about 4 million tokens using 2 GB of main memory and more than 100 million tokens using 64GB main memory (e.g. on a powerful server).

for re-using existing data models and query languages, analyzed problems in the existing relational database implementation and implemented a more flexible approach, which allows to combine different optimization techniques for different kinds of annotations in one single data model and search system. In our mind, especially the concept of specialized Graph Storages for different kinds of annotation layers makes graphANNIS a truly multi-layer corpus search system. Using realistic queries in a benchmark, we could show that graphANNIS clearly outperforms a relational implementation of AQL by a factor of more than 40, and that it leads to speed-up in more than 95% of all tested queries. Being main memory-based, graphANNIS requires a machine proportional to the size of a corpus. All corpora we tested easily fit into 2GB memory, and even a corpus ten times larger than Tiger2 could be queried on a modern desktop computer with 16GB memory.

Besides being very fast, graphANNIS also has a number of other advantages compared to a relational implementation. First, graphANNIS does not require installing and maintaining a database server, a task which has proven very difficult for many linguistic research groups without any IT support. Second, the disk footprint of graphANNIS is much smaller, as none of its internal index structures are stored on disk. This also makes copying corpora much easier and faster; note that corpus import into relANNIS is a rather painful and slow process. Third, starting the AQL server is faster and easier with graphANNIS as no SQL server needs to boot. Finally, for all but the largest corpora the main memory requirements of graphANNIS are actually smaller than those of relANNIS, since the PostgreSQL server itself requires quite a lot of RAM.

Still, there is still room for improvement. For instance, query execution currently is single-threaded, which is a waste of resources on modern computers. Also cost estimates could be improved, and execution plans could be adapted to current CPU technologies (Willhalm et al., 2009). Also data compression has proven very effective in main memory database systems since it leads to much better cache line usage (Abadi et al., 2009). It would be also interesting to benchmark graphANNIS against other linguistic query languages or query systems. For instance, the KorAP system is able to execute AQL queries and if the corpora from our benchmark can be imported into KorAP the existing query set would allow for a good comparison.[15] Additionally to these specialized systems, AQL could be implemented on top of other general purpose main memory database implementations (either relational or graph based) and evaluated if mapping the data model negatively impacts performance.

References

Abadi, D. J., Boncz, P. A., and Harizopoulos, S. (2009). Column-oriented database systems. *Proceedings of the VLDB Endowment*, 2(2):1664–1665.

Ágel, V. and Hennig, M. (2014). Kasseler Junktionskorpus (Version 1.1). Justus-Liebig-Universität Gießen. `http://hdl.handle.net/11022/0000-0000-2102-8`.

[15] A conversion should be possible by using the Pepper framework (Zipser et al., 2010).

Baumann, S. and Riester, A. (2013). Coreference, lexical givenness and prosody in German. *Lingua*, 136:16–37.

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. In Vetulani, Z. and Uszkoreit, H., editors, *Human Language Technology Challenges for Computer Science and Linguistics : 6th language & technology conference december 7-9, 2013, Poznań, Poland*, pages 586 – 587.

Brants, S., Dipper, S., Eisenberg, P., Hansen-Schirra, S., König, E., Lezius, W., Rohrer, C., Smith, G., and Uszkoreit, H. (2004). Tiger: Linguistic interpretation of a german corpus. *Research on Language and Computation*, 2(4):597–620.

Carletta, J., Evert, S., Heid, U., Kilgour, J., Robertson, J., and Voormann, H. (2003). The NITE XML toolkit: Flexible annotation for Multi-modal Language Data. *Behavior Research Methods, Instruments, and Computers*, 35(3):353–363.

Chiarcos, C., Dipper, S., Götze, M., Leser, U., Lüdeling, A., Ritz, J., and Stede, M. (2008). A flexible framework for integrating annotations from different tools and tag sets. *Traitment automatique des langues*, 49(2):271–293.

Coniglio, M., Donhauser, K., and Schlachter, E. (2014). HIPKON: Historisches Predigtenkorpus zum Nachfeld (Version 1.0). Humboldt-Universität zu Berlin. SFB 632 Teilprojekt B4. `http://hdl.handle.net/11022/0000-0000-2D18-4`.

Donhauser, K., Gippert, J., and Lühr, R. (2015). Deutsch Diachron Digital - Referenzkorpus Altdeutsch (Version 0.1). Humboldt-Universität zu Berlin. `http://hdl.handle.net/11022/0000-0000-7FC2-7`.

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.

Färber, F., Cha, S. K., Primsch, J., Bornhövd, C., Sigg, S., and Lehner, W. (2012). Sap hana database: data management for modern business applications. *ACM Sigmod Record*, 40(4):45–51.

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.

Hilpert, M. (2008). *Germanic Future Constructions: A Usage-Based approach to Language Change*. John Benjamins, Amsterdam.

Hirschmann, H., Lüdeling, A., and Zeldes, A. (2008). What's hard? - Quantitative evidence for difficult constructions in German learner data. In *Proceedings of QITL 3. Helsinki*. `http://edoc.hu-berlin.de/docviews/abstract.php?lang=ger&id=37129`.

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. http://dsh.oxfordjournals.org/content/31/1/118.

Lee, J., Yeung, C., Zeldes, A., Reznicek, M., Lüdeling, A., and Webster, J. (2015). Cityu corpus of essay drafts of english language learners: a corpus of textual revision in second language writing. *Language Resources and Evaluation*, 49(3):659–683.

Leech, G. N. (1997). Introducing corpus annotation. In Garside, R., Leech, G. N., and McEnery, T., editors, *Corpus Annotation: Linguistic Information from Computer Text Corpora*, pages 1–18. Longman, London.

Levy, R. and Andrew, G. (2006). Tregex and Tsurgeon: tools for querying and manipulating tree data structures. In *Proceedings of the fifth international conference on Language Resources and Evaluation*, pages 2231–2234.

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

Losemann, K. and Martens, W. (2012). The complexity of evaluating path expressions in SPARQL. In *Proceedings of the 31st symposium on Principles of Database Systems*, pages 101–112. ACM.

Lue, X. (2011). A corpus-based evaluation of syntactic complexity measures as indices of college-level esl writers' language development. *TESOL Quarterly*, 45(1):1–27.

Lüdeling, A. (2011). Corpora in linguistics: Sampling and annotation. In Grandin, K., editor, *Going Digital. Evolutionary and Revolutionary Aspects of Digitization.*, Nobel Symposium 147, pages 220–243. Science History Publications/USA, New York.

Lühr, R., Faßhauer, V., Prutscher, D., and Seidel, H. (2015). Fuerstinnenkorrespondenz 1.1. Universität Jena, DFG. http://hdl.handle.net/11022/0000-0000-82A0-7.

Odebrecht, C. (2011). Lexical Bundles. Eine korpuslinguistische Untersuchung. Master's thesis, Humboldt-Universität zu Berlin. http://edoc.hu-berlin.de/master/odebrecht-carolin-2012-03-12/PDF/odebrecht.pdf.

Odebrecht, C., Belz, M., Zeldes, A., Lüdeling, A., and Krause, T. (accepted 2016). RIDGES Herbology - Designing a Diachronic Multi-Layer Corpus.

Reznicek, M., Lüdeling, A., Krummes, C., Schwantuschke, F., Walter, M., Schmidt, K., Hirschmann, H., and Andreas, T. (2012). Das Falko-Handbuch. Korpusaufbau und Annotationen Version 2.01. Technical report, Technical report, Department of German Studies and Linguistics, Humboldt University, Berlin, Germany. https://www.linguistik.hu-berlin.de/de/institut/professuren/korpuslinguistik/forschung/falko/FalkoHandbuchV2/at_download/file.

Robinson, I., Webber, J., and Eifrem, E. (2013). *Graph Databases*. O'Reilly Media.

Rohde, D. L. (2005). Tgrep2 user manual. http://tedlab.mit.edu/~dr/Tgrep2/tgrep2.pdf.

Rosenfeld, V. (2010). An implementation of the Annis 2 query language. Technical report, Humboldt-Universität zu Berlin. https://www.informatik.hu-berlin.de/de/forschung/gebiete/ti/wbi/teaching/studienDiplomArbeiten/finished/2010/rosenfeld_studienarbeit.pdf.

Rosenfeld, V. (2012). A linguistic query language on top of a column-oriented main-memory database. Master's thesis, Humboldt-Universität zu Berlin. http://www.user.tu-berlin.de/viktor-rosenfeld/assets/publications/diplomarbeit.pdf.

Sauer, S. (2013). BeMaTaC. http://u.hu-berlin.de/bematac.

Schiller, A., Teufel, S., Stöckert, C., and Thielen, C. (1999). Guidelines für das Tagging deutscher Textcorpora mit STTS. Technical report, Universität Stuttgart, Institut fur maschinelle Sprachverarbeitung; Universität Tübingen, Seminar für Sprachwissenschaft.

Seufert, S., Anand, A., Bedathur, S., and Weikum, G. (2013). Ferrari: Flexible and efficient reachability range assignment for graph indexing. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 1009–1020. IEEE.

Stede, M. and Neumann, A. (2014). Potsdam Commentary Corpus 2.0: Annotation for Discourse Research. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*, Reykjavik, Iceland. European Language Resources Association (ELRA). http://www.lrec-conf.org/proceedings/lrec2014/pdf/579_Paper.pdf.

Telljohann, H., Hinrichs, E., Kübler, S., Zinsmeister, H., and Beck, K. (2009a). Stylebook for the Tübingen Treebank of Written German (TüBa-D/Z). Technical report, Universität Tübingen Seminar für Sprachwissenschaft. http://www.sfs.uni-tuebingen.de/ascl/ressourcen/corpora/tueba-dz.html.

Telljohann, H., Hinrichs, E. W., Kübler, S., Zinsmeister, H., and Beck, K. (2009b). *Stylebook for the Tübingen Treebank of Written German (TüBa-D/Z)*. Universitüt Tübingen Seminar für Sprachwissenschaft, Wilhelmstr. 19 D-72074 Tübingen.

Walter, M. (2015). Märchenkorpus Version 1.0. Humboldt-Universität zu Berlin. http://www.textbewegung.de/. http://hdl.handle.net/11022/0000-0000-8211-9.

Willhalm, T., Popovici, N., Boshmaf, Y., Plattner, H., Zeier, A., and Schaffner, J. (2009). SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. *Proceedings of the VLDB Endowment*, 2(1):385–394.

Wood, P. T. (2012). Query languages for graph databases. *ACM SIGMOD Record*, 41(1):50–60.

Yildirim, H., Chaoji, V., and Zaki, M. J. (2010). Grail: Scalable reachability index for large graphs. *Proceedings of the VLDB Endowment*, 3(1-2):276–284.

Zeldes, A. (2016a). *ANNIS User Guide - Version 3.4.3*. http://corpus-tools.org/annis/resources/ANNIS_User_Guide_3.4.3.pdf.

Zeldes, A. (2016b). The GUM corpus: creating multilayer resources in the classroom. *Language Resources and Evaluation*, pages 1–32.

Zhang, H., Chen, G., Ooi, B. C., Tan, K.-L., and Zhang, M. (2015). In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1920–1948.

Zinsmeister, H., Reznicek, M., Brede, J. R., Rosén, C., and Skiba, D. (2012). Das Wissenschaftliche Netzwerk „Kobalt-DaF". *Zeitschrift für germanistische Linguistik*, 40(3):457–458. https://dx.doi.org/10.1515/zgl-2012-0030.

Zipser, F., Romary, L., et al. (2010). A model oriented approach to the mapping of annotation formats using standards. In *Workshop on Language Resource and Language Technology Standards, LREC 2010*.