A Linguistic Query Language On Top Of A Column-Oriented Main-Memory Database

Diplomarbeit

zur Erlangung des akademischen Grades
Diplominformatiker

Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät II
Institut für Informatik

eingereicht von:	Viktor Rosenfeld
geboren am:	24. Februar 1980
in:	Berlin
Gutachter:	Prof. Dr. Ulf Leser
	Dr. Stefan Manegold
eingereicht am:	. . . . . .

Statement of authorship

I declare that I completed this thesis on my own and that information which has been directly or indirectly taken from other sources has been noted as such. Neither this nor a similar work has been presented to an examination committee.

Berlin, September 4, 2012 ....

Contents

List of Tables

List of Figures

List of Listings

1 Introduction

Linguists rely on corpora of actual spoken or written language, such as newspaper articles, to study language variety. Natural language processing algorithms, e.g., automated part-of-speech tagging or syntax tree generation, have allowed the costruction of large corpora, containing millions of words, which are further enriched with a multitude of data [Lüd11]. The challenge posed by these corpora is to quickly identify and retrieve examples of a linguistic phenomenon a researcher is interested in. To satisfy this demand, we have developed Annis, a database and web-based multi-layer corpus system [ZRLC09]. It provides a simple, yet expressive query language which is translated to SQL and evaluated on PostgreSQL [Pos96].

The current implementation of Annis is able to evaluate complex linguistic queries on multi-layer corpora containing hundreds of thousands of words at a speed which is suitable for interactive use [Ros11]. This performance is, to the best of our knowledge, unparalleled by other multi-layer corpus systems. However, Annis reaches its limits with corpora containing close to a million words. It makes extensive use of denormalization and indexes which unfortunately results in a very large disk footprint and limits the extensibility of the language. In order to alleviate these disadvantages, we have developed a prototype implementation of the Annis query language on top of MonetDB, a column-oriented database system suited for data-intensive, analytical workloads [BK99].

In this work, we describe the necessary steps to implement Annis on MonetDB, measure the performance of the port, and compare it with the current implementation on PostgreSQL. Specifically, we want to evaluate Annis queries on the normalized database schema so as to retain its flexibility. In order to use a realistic workload in our evaluation, we collect queries from an Annis installation at the linguistics department of the Humboldt-Universität zu Berlin.

In the remainder of this section, we provide a background on the use of annotated corpora in linguistics and discuss why it is expedient to implement a linguistic query language on top of a database. We then give an overview of the specific requirements and history of Annis and discuss why a column-oriented database is a suitable basis for Annis.

1.1 Corpora as resources for linguistic study

Corpus linguistics is a branch of linguistics that emphasizes the study of actual language use by analyzing large collections of texts as empirical evidence.[1] In particular, corpus linguists are interested in language variation: What options does a speaker or writer have when expressing themselves and what internal or external factors influence their choices.

One example is the flexibility a German speaker has with regard to word order. German syntax as described by the Stellungsfeldermodell [Dra37] (see also [ZHS+97]) uses the position of the finite verb as a fixed point in the structure of the sentence. In a *V2 sentence* or *Verbzweitsatz*, in which the finite verb appears in the second position, the content of the first position, called the *prefield*, is underspecified by syntax. Specifically, the subject is no more privileged by syntactical considerations than other constituents. Compare this to English grammar which generally conforms to a fixed *subject-verb* order, although it is inverted in some circumstances [BJL+99]. This is demonstrated in by the three sentences in Figure 1. The first sentence is taken from the Potsdam Commentary Corpus [Ste04]. The finite verb at the second position makes up the *left sentence bracket* (German: linke Satzklammer, LK). A German sentence also contains a *right sentence bracket* (rechte Satzklammer, RK), however in this example it is empty. The *prefield* (Vorfeld, VF) of the sentence precedes the left sentence bracket and typically consists of exactly one phrase. Between the left and right sentence brackets lies the *middle field* (Mittelfeld, MF). It can contain multiple constituents which are underlined in the example. The next two sentences

[1]There is a considerable debate among corpus linguists about what exactly is meant by the term *corpus linguistics*. In [Tay08], Taylor cites many definitions offered within research of corpus linguistics ranging from a very narrow focus as a tool or methodology to very broad definitions as a linguistic discipline or paradigm. For an introduction into corpus linguistics see, e.g., [BCR98].

VF	*LK*	*MF*	*RK*
Das	forderten	sie bei der ersten Zossener Runde am Dienstagabend.	
Sie	forderten	das bei der ersten Zossener Runde am Dienstagabend.	
Am Dienstagabend	forderten	sie das bei der ersten Zossener Runde.	

Figure 1: Three German sentences consisting of the same constituents in a different word order.

demonstrate how phrases from the middle field can be moved into the prefield without changing the meaning of the sentence.

Given that a speaker is relatively free to choose which constituent to put in the prefield, a linguist may want to ask what purpose is served by this flexibility. To this end he may study which constituents are possible in the prefield, what their distribution is, and if there is a clear preference for one type [SJ08]. He may want to investigate the role played by the constituents in the prefield with regard to the discourse structure of the text [Spe08] and if there is a difference between written and spoken language [Spe10]. It may also be interesting to compare German with a closely related language such as Swedish [BR08] or a relatively dissimilar language such as Chinese [BZ10] and investigate possible reasons for unidiomatic language use by non-native speakers. Such an analysis must be based on a large amount of language in order to limit the influence of a few speakers' or writers' idiosyncrasies. Nevertheless, it is typically impossible to analyze a linguistic variety of a natural language in its entirety. The number of texts or utterances in a modern language is practically infinite. A corpus linguist therefore has to create a sample of the linguistic variety he is interested in, i.e., construct a *corpus*. This sample corpus is then used as an empirical basis for research. Some quantitative research questions can be answered by only using the text itself. For example, Rayson et al. analyzed the spoken component of the British National Corpus [Cro93] in order to find significant deviations in word usage frequency in groups differentiated by gender, age, and social group [RLH97].

However in most cases, the usefulness of a corpus is greatly enhanced if it is enriched with additional information derived from the text. If this information is stored within the corpus, it is called an *annotation*. On a macroscopic level, annotations can contain general information about a text such as the name, age or gender of the author, when the text was written or published, and what language it is in. On the other side of the spectrum are positional annotations which describe the smallest units of the text. Typically, each word is classified according to morphological and lexical criteria. Examples of positional annotations are the lemma, i.e., the word's canonical form, and part-of-speech information, i.e., its syntactic word class. Finally, there are structural annotations which describe the logical structure of the text. Examples are syntax trees, i.e., the internal structure of individual phrases, dependency trees, i.e., the semantic arguments of individual words, or co-reference links, i.e., assignments of different phrases in a text to the same entity in real life. Figure 2 shows a fragment of the Potsdam Commentary Corpus annotated with multiple positional and structural annotations. Below the text are three rows of token annotations, the morphological and part-of-speech category as well as the lemma of the token. Further below are two annotation layers that span multiple tokens. The first contains information about the cognitive status of the phrase and the second whether a phrase contains new information or refers to an earlier part of the text. Printed above the text is a syntax tree consisting of black lines in which nodes are labeled with the syntactic category and edges are labeled with the function of the constituent. Finally, there are two annotation layers implemented as labeled links: A dependency tree linking verbs and nouns with their arguments using grey dashed lines and a coreference link connecting two phrases that refer to the same entity with a green line.

Annotations typically do not add new information to a corpus, but make explicit what is already implicitly included in a text. However, their presence is a requirement for the efficient use of corpora in many cases. Instead of looking at every sentence in a corpus to find those in which the subject follows the verb, in a syntactically annotated corpus one can search for that data directly. For an in-depth discussion of corpora and their use in corpus linguistics, literary computing, and computational linguistic see [LZ08] and [Lüd11].

In recent years, richly annotated corpora have been constructed which go beyond simple syntactical

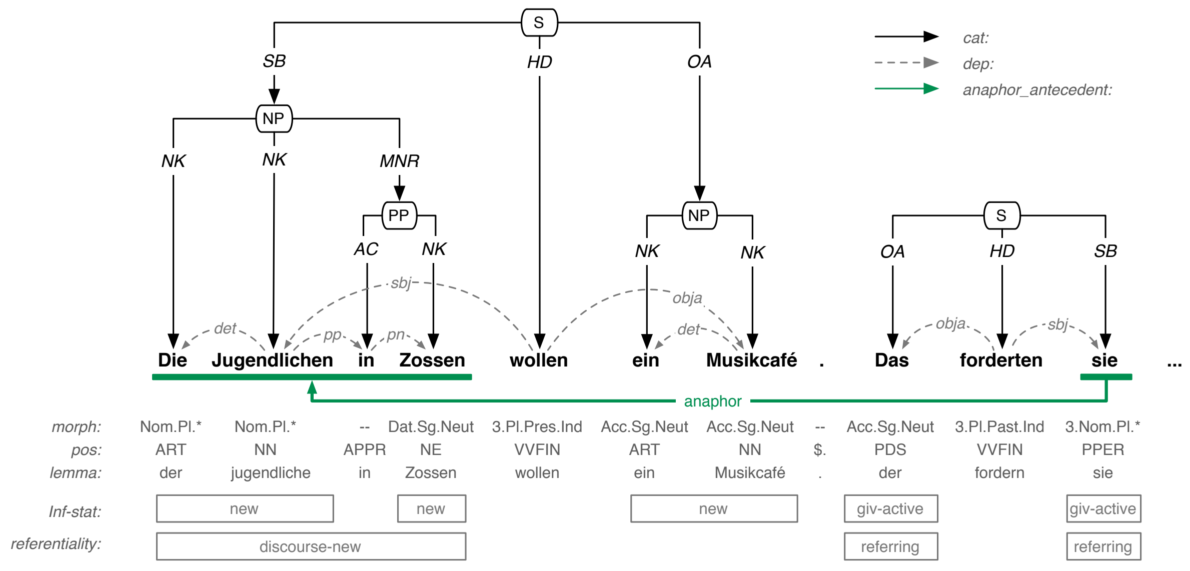


Figure 2: A fragment from the Potsdam Commentary Corpus with annotations.

annotations and integrate annotations from a wide variety of linguistic areas. Such corpora are called *multi-layer* or *multi-level corpora*. For example, Pustejovsky et al. [PMPP05] merged various complementary annotations of *Wall Street Journal* articles from the Penn Treebank [MMS93], namely PropBank [PGK05] and NomBank [MRM04] which contain predicate argument structures anchored on verbs and nouns respectively, TimeBank [PIS+05] containing temporal features of propositions and the temporal relations between propositions, the Penn Discourse Treebank [PDL+08] linking discourse connectives with the sentences they join, and coreference annotations linking text segments referring to the same entities [PV98]. The Potsdam Commentary Corpus [Ste04] is a similarly annotated corpus of German regional newspaper texts and the OntoNotes corpus [WHM+11] contains texts from the Penn Treebank as well as Chinese and Arabic news texts. Falko [LDH+08] and CzeSL [HRŠŠ10] are error-tagged learner corpora of non-native German and Czech speakers respectively. They are examples of corpora in which annotations add genuine new information, specifically an interpretation of the speakers intended meaning.

These corpora are particularly valuable because they allow the study of complex phenomena which operate on several linguistic levels simultaneously. However, they pose special demands on corpus architectures, first and foremost the ability to model overlapping and concurrent hierarchical annotations of the same text. These are also called *conflicting* annotations because they typically cannot be expressed in a well-formed nested structure, such as an XML tree. An interesting side effect is the ability to store *competing* annotations in one corpus, i.e., multiple annotations that are semantically similar, but motivated by different theoretical frameworks or annotation guidelines. Multi-layer corpus architectures also need to be able to integrate the output of established tools in order to support historically available data and established workflows.

Given the complexity and size of multi-layer corpora it is not feasible to simply browse them. Instead, users must be able to directly search a corpus for features they are interested in. A corpus query language must first provide the means to express relationships between linguistic phenomena, e.g., their position in a syntax tree or with regard to the linear order of a text. A query language for multi-layer corpora also needs to be able to distinguish and access concurrent hierarchies and other overlapping annotations. On the one hand, it must be sufficiently expressive in order to enable the user to precisely state the phenomena they are interested in, otherwise the search will yield many unwanted results. On the other hand, it must also be accessible to users who may have little or no programming experience. Finally, it must be able to quickly search large corpora. Corpus data is often stored in XML files. Attempts to operate directly on these files, using custom data structures or the help of an XML database, have not yielded a satisfactory performance [DGSW04, RSW+09]. We have approached this problem by defining a domain-specific language that is translated to SQL and evaluated on a relational database management system, or RDBMS. The efficient evaluation of queries on large datasets has been studied extensively by the database community [AAB+08]. By using a database as a query processor we can benefit directly from this work.

1.2 Annis – A multi-layer corpus architecture

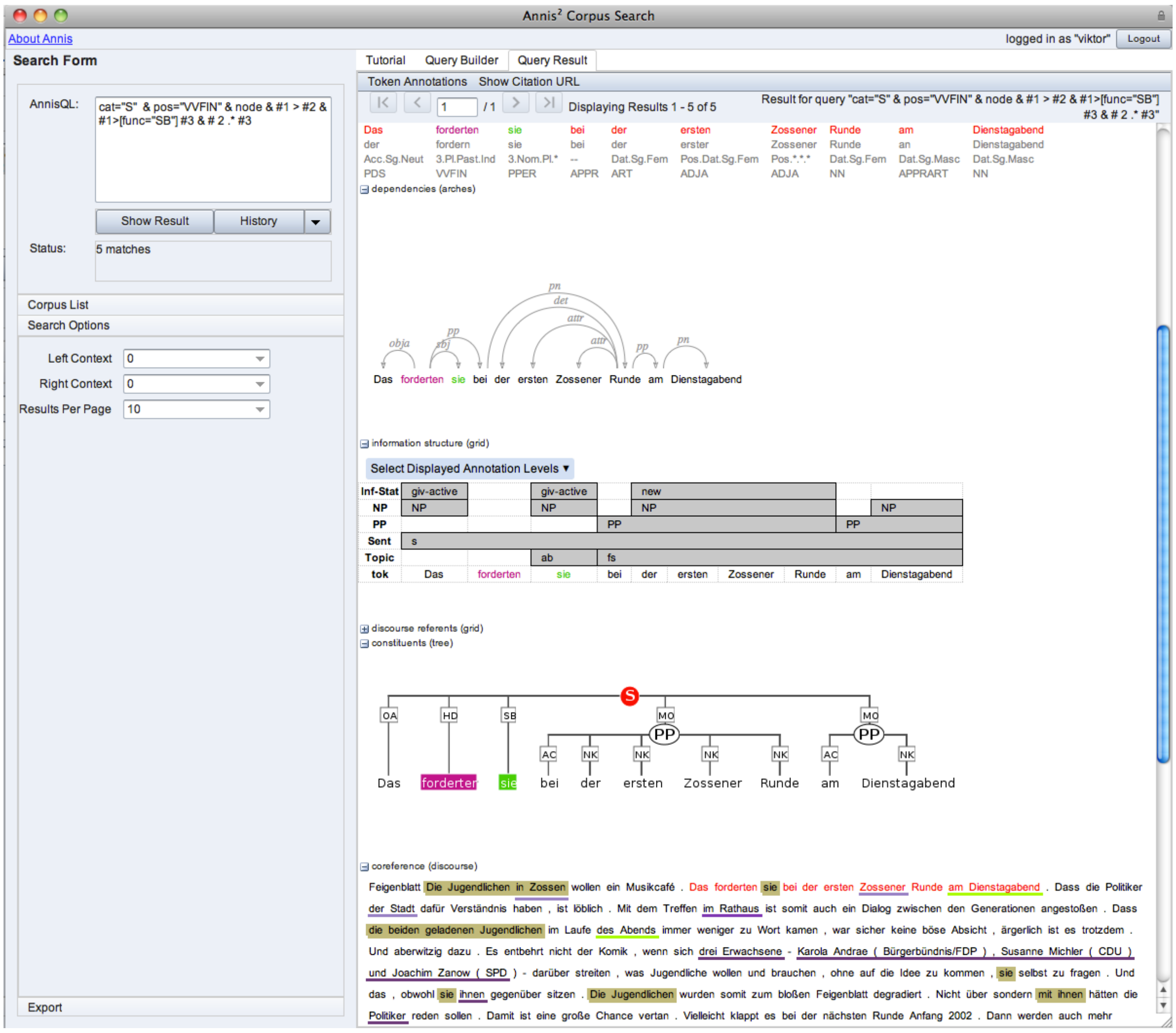
Annis[2] is a database and web-based multi-layer corpus system developed within the Collaborative Research Center SFB 632 on Information Structure[3] in cooperation between the University of Potsdam and the Humboldt Universität zu Berlin. The underlying assumption of the study of information structure is that the same information is "packaged" differently, depending on the context and the goal of the discourse. The SFB brings together researchers from linguistics, psychology, and African studies who investigate the interaction of information structure with various linguistic aspects, e.g., the choice of lexical means and the composition of texts, as well as the cognitive processing of information. It uses empirical data in a wide variety of languages and many different annotation schemes. Consequently, the SFB requires tools that are able to integrate diverse, possibly conflicting annotations of the same text, such as the example shown in Figure 2, and to enable researchers to access different annotation layers simultaneously. Annis is the result of an effort to provide a system meeting these requirements. Exemplary studies showcasing how to use Annis with multi-layer corpora are [ZRLC09, CRS11], and [KRZZ11].

A central focus of Annis is the suitable visualization of different annotation schemes. Some of the possibilities are showcased in the screenshot of Annis shown in Figure 3. The large area on the right side of the screenshot contains different visualizations of the annotations of a sentence from the Potsdam Commentary Corpus. At the top, the sentence is displayed along with lemma, part-of-speech, and morphology annotations underneath each token. The graph below is a dependency tree rooted in the verb of the sentence. Annotations across multiple tokens are shown in an annotation scheme-specific grid view in the middle, followed by a second hierarchical annotation, the syntax tree of the sentence. Finally, the entire text is shown in a discourse view in which coreferential text segments are underlined with the same color and phrases referring to *die Jugendlichen in Zossen* are highlighted. Annis also supports multi-media annotations which can be aligned with individual text spans. The system is fully Unicode-compliant and supports left-to-right as well as right-to-left languages.

The original version of Annis, now called Annis 1, operated on in-memory data structures which were generated at startup from XML files containing the available corpora and their annotations. Annis 1 did not directly support tree visualizations; instead trees were generated with an external tool and stored as graphical annotations. It soon became apparent that the in-memory architecture of Annis 1 severely constrained the amount of data the system could process on then current hardware [DGSW04]. Thus, the next version, Annis 2, was implemented on top of the open source database system PostgreSQL [Ros11]. The system was extended with direct support for the visualization of hierarchical annotations and a graphical query builder [Hüt08]. Additionally, a formal framework was developed for the conversion of various linguistic storage formats [ZR10] which is used to load corpus data into Annis.

The main optimization strategy in [Ros11] was to denormalize most of the Annis database schema into a single facts table and to build multiple, combined and partial B-Tree indexes dedicated to the implementation of specific language features. This strategy enabled us to evaluate many queries interactively on modern consumer hardware. However, it also has significant drawbacks. The large number of indexes greatly increases the space required to store a corpus. Consequently, if the corpus does not fit into main memory, disk I/O is also increased during query processing. Denormalization also limits the flexibility of the database schema. Adding more attributes, e.g., to implement a new language feature, increases the width of the facts table and with it the potential I/O load of every query, even those that do not use the new feature.

[2]http://www.sfb632.uni-potsdam.de/d1/annis/
[3]http://www.sfb632.uni-potsdam.de/



AnnisQL: cat="S" & pos="VVFIN" & node & #1 > #2 & #1>[func="SB"] #3 & # 2 .* #3

Status: 5 matches

Left Context	0
Right Context	0
Results Per Page	10

1 / 1 Displaying Results 1 - 5 of 5

Result for query "cat="S" & pos="VVFIN" & node & #1 > #2 & #1>[func="SB"] #3 & # 2 .* #3"

Das	forderten	sie	bei	der	ersten	Zossener	Runde	am	Dienstagabend
der	fordern	sie	bei	der	erster	Zossener	Runde	an	Dienstagabend
Acc.Sg.Neut	3.Pl.Past.Ind	3.Nom.Pl.*	--	Dat.Sg.Fem	Pos.Dat.Sg.Fem	Pos.*.*.*	Dat.Sg.Fem	Dat.Sg.Masc	Dat.Sg.Masc
PDS	VVFIN	PPER	APPR	ART	ADJA	ADJA	NN	APPRART	NN

Inf-Stat	giv-active		giv-active		new					
NP	NP		NP		NP					NP
PP				PP					PP	
Sent	s									
Topic			ab	fs						
tok	Das	forderten	sie	bei	der	ersten	Zossener	Runde	am	Dienstagabend

Das forderten sie bei der ersten Zossener Runde am Dienstagabend

Feigenblatt Die Jugendlichen in Zossen wollen ein Musikcafé . Das forderten sie bei der ersten Zossener Runde am Dienstagabend . Dass die Politiker der Stadt dafür Verständnis haben , ist löblich . Mit dem Treffen im Rathaus ist somit auch ein Dialog zwischen den Generationen angestoßen . Dass die beiden geladenen Jugendlichen im Laufe des Abends immer weniger zu Wort kamen , war sicher keine böse Absicht , ärgerlich ist es trotzdem . Und aberwitzig dazu . Es entbehrt nicht der Komik , wenn sich drei Erwachsene - Karola Andrae (Bürgerbündnis/FDP) , Susanne Michler (CDU) und Joachim Zanow (SPD) - darüber streiten , was Jugendliche wollen und brauchen , ohne auf die Idee zu kommen , sie selbst zu fragen . Und das , obwohl sie ihnen gegenüber sitzen . Die Jugendlichen wurden somit zum bloßen Feigenblatt degradiert . Nicht über sondern mit ihnen hätten die Politiker reden sollen . Damit ist eine große Chance vertan . Vielleicht klappt es bei der nächsten Runde Anfang 2002 . Dann werden auch mehr

Figure 3: Screenshot of Annis displaying a sentence and its annotations.

1.3 Main-memory and column-oriented database systems

As the size of available main memory increases, it has become possible to keep entire databases, or at least the hot set of very large databases, resident in main memory [GMS92]. Thus, the overriding optimization criteria for a traditional RDBMS, the reduction of disk I/O, has become less important and other optimization criteria have become relevant. Although main memory is randomly accessible, from the point of view of a modern processor, not all of it is accessible at the same speed. Access to memory regions that have not yet been loaded into the CPU cache will incur a cache miss penalty during which the processor effectively stalls if it cannot find work to do in parallel. Indeed, many commercial RDBMSs spend much of the query execution time on main memory and other resource stalls [ADHW99]. To achieve optimal performance on modern processors, it is necessary to improve the cache behavior of data structures and query processing algorithms [BMK99]. These differences impact all components of an RDBMS [MKB09].

Traditional database systems employ an n-ary storage model, or NSM. They store the values of every attribute of an individual tuple together. Using a table-based metaphor for a relation, traditional databases store individual rows. In contrast, column-oriented database systems, also called column-stores, implement a decomposed storage model, or DSM. They store the values of a single attribute from every tuple together. A relation with n attributes is vertically partitioned into n binary attribute relations and the attribute values of a tuple are linked by surrogate keys.

The benefits of the DSM were first described in [CK85]. Queries that only reference some of the attributes of a database schema can be evaluated faster on the DSM than on the NSM. Because a column-store can load only the attributes required to evaluate a query, it reduces disk I/O and makes better use of the buffer cache. This rationale not only applies to the reduction of disk I/O, but also to other parts of the memory hierarchy. Indeed, a vertically partitioned data structure optimizes CPU cache usage [BMK99]. These theoretical advantages have been realized in a number of data-intensive, OLAP applications, including Data Mining [BRK98], Data Warehouses [SBÇ+07], and Scientific Computing [INGK07]. In these fields, column-stores can achieve an advantage over traditional RDBMS by an order of magnitude or more.

Annis uses the database in a similar way as other OLAP applications. Annis queries often access only some attributes of the database schema, but need to process many or all of the values of a particular attribute to obtain a result. A corpus is not modified once it is loaded into Annis, i.e., the database is used in a read-only fashion. However, Annis does not fit the OLAP model perfectly. Except for simple queries, Annis joins the central table of the database schema with itself many times over. Furthermore, a fairly typical query does indeed, for a small number of tuples, access almost every attribute of the database schema. A column-store is ill-suited for this use case because its performance degrades with the number of projected attributes. Nevertheless, we think that an implementation of Annis on a column-store can eliminate the disadvantages of the current implementation on PostgreSQL. Specifically, it should be able to evaluate Annis queries on the normalized database schema.

In the last decade many column-stores have been developed [ABH09]. We have chosen MonetDB as the target of the port [BK99]. MonetDB pioneered the research into column-oriented, main-memory databases in the 1990s and is actively developed and backed by a strong research community. At its core, MonetDB implements an algebra on binary association tables, or BATs. A BAT stores the values of an individual relational attribute. BATs are manipulated using the MonetDB Assembly Language, or MAL. In order to answer a SQL query, it is translated into a MAL program, which is then executed by the MonetDB kernel.

1.4 Structure of this work

In section 2, we explain the Annis data model and its query language in detail, followed by a description of the translation process of Annis queries to SQL, and the current implementation on PostgreSQL in section 3. We discuss other linguistic query languages and compare them with Annis in section 4. In section 5, we describe the TIGER Treebank, the test queries we use to measure the performance of Annis, and other aspects of our testing environment. The implementation of Annis on MonetDB is described and evaluated in section 6. We conclude our work in section 7. In appendix A, we describe Annis features that we did not port to MonetDB. Appendix B contains performance data of the test queries.

2 The Annis corpus system

This section summarizes the current state of Annis. We describe how data is prepared for import into Annis, explain the data model that is used to store this data, and give an overview of the Annis query language.

2.1 System architecture

Figure 4 shows an overview of the Annis system architecture. The user interacts with the system primarily through a web browser and the specialized Annis query language (AQL). Queries can be constructed with a graphical query builder or entered as text. They are translated by a back-end service into SQL which is then executed on an RDMBS. The returned data is converted into Annis data structures which are presented to the user with the help of dedicated visualizers. Alternatively, the data returned by the Annis service can be exported for further analysis in external tools such as WEKA [Gar95] and R [R11].

Annis itself does not have any facilities to create annotations. Instead, it operates on pre-annotated data that may have been generated by multiple annotation tools and merged into a unified data structure before being imported into Annis. This approach allows users to leverage specialized tools which are purpose-built for the creation of specific annotation schemes. The process is summarized by the diagram in Figure 5. *Primary data* is a collection of texts. These texts are annotated with the help of external tools either automatically or by a user. The output is then transformed by the Salt'N'Pepper [ZR10] converter framework into a set of relANNIS files before being imported into an RDBMS.

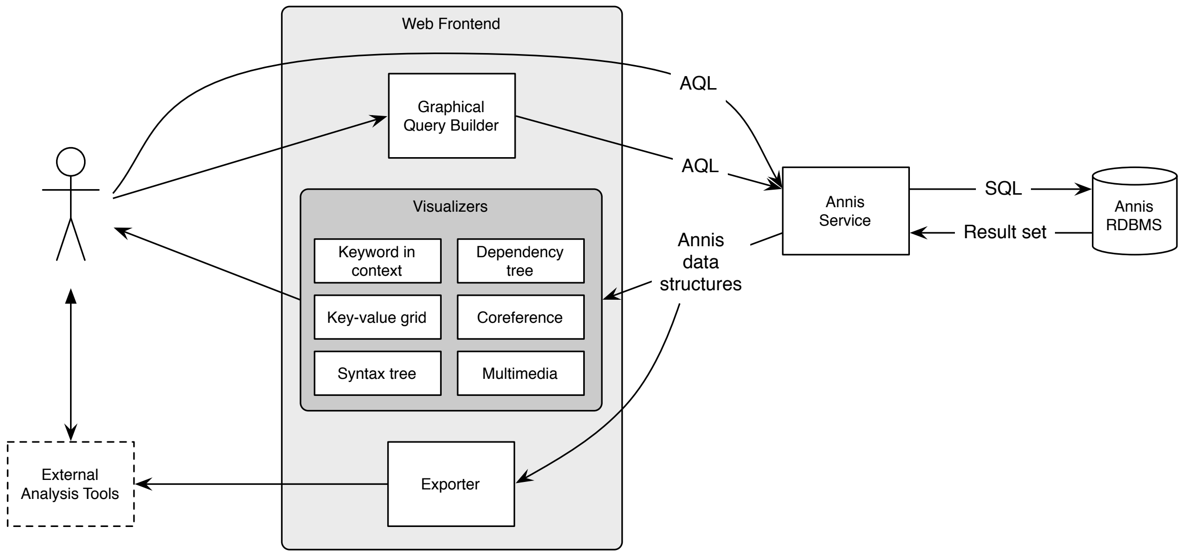


Figure 4: User interaction and data flow in Annis.

2.2 Data model

The Annis corpus system can store many corpora and allows the evaluation of an Annis query on multiple corpora simultaneously. Corpora are arranged in a corpus hierarchy. The contents of a single corpus is stored in an annotation graph.

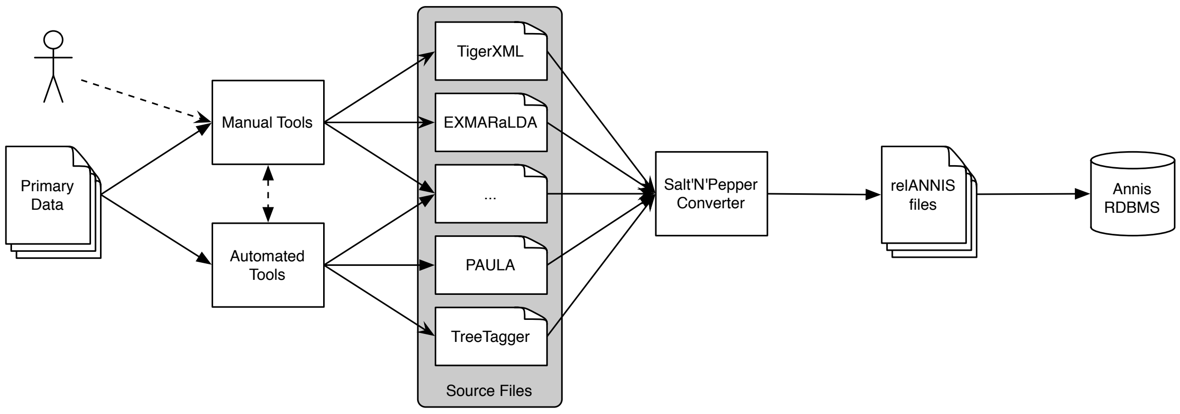


Figure 5: Creation of a multi-layer linguistic corpus for Annis.

2.2.1 The annotation graph

An *annotation graph*[4] is an ordered directed acyclic graph, or ODAG, which integrates the annotations across multiple annotation layers of a corpus. The nodes in this graph are *text spans*, i.e., substrings of a particular primary text, and the edges encode linguistic relationships between these spans. A corpus can contain multiple texts simultaneously and a linguistic relationship may connect text spans from different texts.

Each node has three main properties: the left and right index (inclusive) of the text span in the text and a reference to that text. Nodes are implicitly nested if a text span is contained within another. However, text spans may also overlap arbitrarily and it may not be possible to arrange the nodes of the graph in a tree structure. A subset of the nodes, usually but not necessarily the leaves of the graph, are the tokens of text. They have two additional properties: the token's textual content and its position in the text. A node can be annotated with arbitrary key-value pairs to encode a positional annotation of a token or a structural annotation of a non-token text span. There can be multiple nodes covering the same text span, but annotated with different key-value pairs, to model independent annotation layers.

Each edge is typed to distinguish different semantics of the parent-child relationship. Three different edge types are currently supported:

- *Coverage* edges group one or more child spans. The child spans do not have to be continuous. This allows the construction of text spans containing gaps, such as the verb phrase in Figure 15 on page 37.
- *Pointing relation* edges are used to encode arbitrary, possibly non-hierarchical, relationships between text spans. For example, they are used to model dependency relationships between verbs or nouns and their arguments. Note that a pointing relation typically does not imply coverage.
- *Dominance* edges encode the dominance hierarchy of text spans, i.e., a syntax tree. Note that dominance typically implies coverage but not vice versa. A dedicated hierarchical dominance edge type, in addition to the more general pointing relation type, is used to provide dedicated dominance operators in the Annis query language.

Other linguistic relationships could be modeled using a dedicated edge type if this were to improve the expressiveness of the query language. In addition to a type, each edge has an optional name to distinguish different subtypes. For pointing relation edges this name is used to model different kinds of linguistic relationships. Similarly to nodes, an edge can be annotated with arbitrary key-value pairs. There may be multiple edges of the same type between the same two nodes, possibly annotated with different key-value pairs, to model independent annotation layers.

[4]Note that our usage of the term annotation graph differs from Lai and Bird in [LB04].

In the model described above there are three different ways to express a relationship between two text spans:

- By an explicit relationship in the graph, i.e., the existence of an edge or a path between the spans.
- By a precedence relationship, which compares the position of the left-most and right-most tokens covered by the spans.
- By a coverage relationship, which compares the left and right text borders of the spans.

These relationships are demonstrated in Figure 6. The graph encodes a phrase and multiple annotation layers from the Potsdam Commentary Corpus. Solid boxes denote the text spans of the annotation graph and dotted boxes denote node or edge annotations. At the bottom are four tokens, each with a *pos*, *morph*, and *lemma* annotation. The numbers in the top-left corners represent the token position in the text and the numbers in the bottom corners the left and right index of the text spans. Above are four different annotation layers represented by shaded boxes. The annotation layer at the top encodes a syntax tree using annotated nodes and green dominance edges. The next two boxes denote two separate non-hierarchical annotation layers using annotated nodes. They are linked to the constituent tokens using dashed coverage edges. The last annotation layer encodes a dependency tree using blue pointing relation edges. In this graph, the nodes A and B both cover the token D, but only A dominates D. The token C precedes the token D and is linked to it by a pointing relation. The token C also precedes the node B because B's left-most and right-most covered token is D.

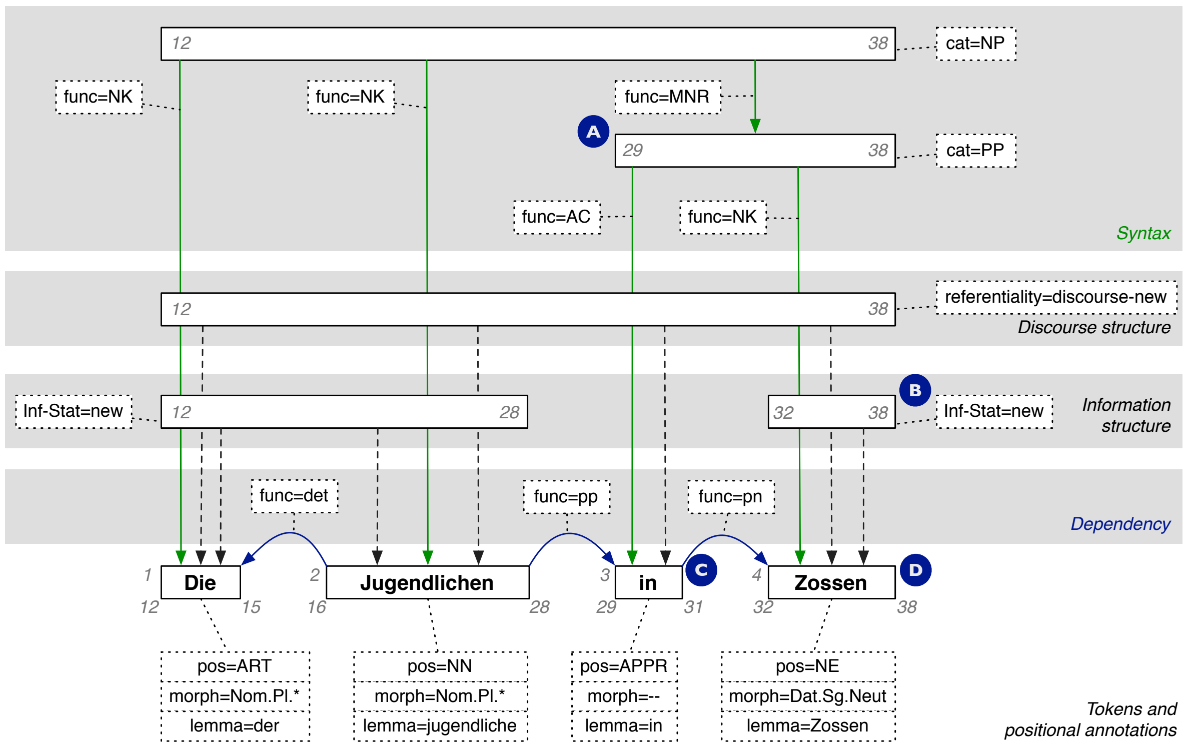


Figure 6: Annotation graph modeling a phrase from the PCC corpus and some of its annotations.

2.2.2 The corpus hierarchy

A linguistic corpus is often divided into subcorpora by its authors. In Annis, this relationship is modeled by a corpus hierarchy. We use the term *corpus*, or *root corpus*, to refer to a corpus at the root of a corpus hierarchy. A subcorpus is also called a *document*. Documents can be annotated with arbitrary key-value

```
1  cat="S" & #1:root &            match full sentences
2  pos="VVFIN" & #1 > #2 &        and their finite verb
3  node & #1 >[func="SB"] #3 &    and their subject
4  #2 .* #3 &                     where the verb precedes the subject
5  meta::Genre="Politik"          only consider documents of the genre Politik
```

Listing 1: Annis query matching sentences in which the subject follows the verb.

pairs to store meta data. Conceptually, there is no difference between a corpus and document, except how they are accessed by the user. The web interface presents a list of root corpora from which the user selects one or more entries. A query is simultaneously evaluated on the selected corpora and their child documents. The user can limit the evaluation of the query to documents annotated with specific meta data but cannot select documents directly.

2.3 The Annis query language

An Annis query defines a subgraph template that is matched against the annotation graphs of one or more corpora. There are three types of predicates:

- *Search terms* select tokens by their textual content or any node by a key-value annotation. They are implicitly numbered by Annis in the order they appear in the query, starting with 1.
- *Unary linguistic constraints* filter nodes by a node-specific property, such as the number of tokens that are covered by the node.
- *Binary constraints* enforce a linguistic relationship between two nodes, such as precedence or dominance.

Unary and binary constraints refer to search terms by their implicit index prefixed with a hash mark, e.g., `#1` for the first search term listed in the query. Predicates are combined using `&` (AND) and `|` (OR) and can be grouped with parentheses. The search can be restricted to documents annotated with certain meta information.

Listing 1 contains an example of an Annis query. A matching annotation graph from the PCC corpus is depicted in Figure 7. Lines 1 through 3 define three search terms and their position in a dominance hierarchy: A full sentence (`cat="S"`) dominating a finite verb (`pos="VVFIN"`) and a text span via an edge that denotes that the constituent serves as the subject of the sentence (`func="SB"`). In line 4 the relative position of the verb and the subject is specified. Finally, line 5 restricts the search to documents of a specific genre. Figure 7 depicts a matching dominance hierarchy from the PCC corpus. The labels 1, 2, and 3 indicate the nodes matched by the respective search term of the query. Only matched annotations are depicted, while unmatched nodes and edges are grayed out. Note that annotation names, e.g., `cat`, `func`, or `Genre` in Listing 1, and annotation values, e.g., `"VVFIN"` or `"SB"`, are specific to a corpus and/or an annotation scheme. In another corpus, features such as sentences, verbs, or the genre of the document may be encoded using different keywords or not at all. Actual Annis keywords, e.g., **`node`**, are printed with a boldface font.

2.3.1 Query functions

Annis queries typically are not evaluated directly but used as a building block for a *query function*. A query function first computes the *solutions* of the Annis query. A query solution, or match, is an assignment of nodes to the query's search terms, so that the annotation subgraph induced by these nodes matches the template defined by the query. In a second step, the query function retrieves information related to these solutions from the corpus. Currently, there are three implemented query functions:

- *COUNT* counts the number of subgraphs matching an Annis query.

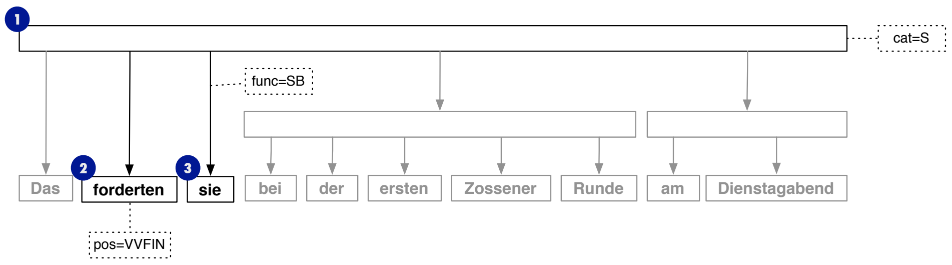


Figure 7: Annotation graph of the dominance hierarchy matched by the Annis query in Listing 1.

- *ANNOTATE* retrieves for each matched subgraph any text span covering a span of the subgraph, optionally with a left and right context. The result of this function is transformed by the web front-end to visualize the annotations of the text surrounding a match, i.e., to create the view on the right side of the screenshot shown in Figure 3 on page 12. Because the amount of data returned by this function grows quickly there exists a paged variant as well.
- *MATRIX* returns a table which for each matched subgraph lists the annotation values of the spans in the graph. The result of this function is exported as an ARFF file [Gar95] for further analysis by external tools.

2.3.2 Search terms

There are three types of search terms:

- The keyword **`node`** selects any span in the database. Similarly, the keyword **`tok`** selects any token.
- *Text search*: `"Mary"`, or `tok="Mary"`, selects tokens by exactly matching their textual content. Alternatively, `/Mar(y|ie)/`, or `tok=/Mar(y|ie)`, matches the text of a token against a regular expression. Regular expressions are implicitly anchored by Annis. In the example, the regular expression `^Mar(y|ie)$` would be evaluated. It is also possible to select tokens not matching a string or regular expression with `tok!="Mary"` or `tok!=/Mar(y|ie)/`.
- *Annotation search*: `pos="VVFIN"` selects text spans by their annotation. `pos!="VVFIN"` selects nodes with a `pos` annotation that is different from `"VVFIN"`. The annotation value may also be specified using a regular expression. If the value is omitted, any span with the given key will be matched. Annis does not make any assumptions about the meaning of the annotation name `pos` or the value `"VVFIN"`. These strings are specified by the corpus annotation scheme and are not Annis keywords.

Search terms are implicitly numbered by Annis in the order they appear in the query, starting with 1. This index is used by linguistic constraints to refer to a particular search term.

2.3.3 Unary linguistic constraints

Unary linguistic constraints follow the form "`#i:condition`" where i is a search term reference. There are three unary constraints, listed in Table 1. Note that `#i:root` matches root nodes in the original annotation graph and not those nodes that are the root of one connected component, but a leaf in another. See section 3.1.2 for details.

2.3.4 Binary linguistic constraints

Binary linguistic constraints follow the form "`#i operator #j`", where i and j are search term references. There are four families of operators:

Table 1: Unary linguistic constraints in AQL.

Operator	Definition
`#i:root`	i is a root node
`#i:arity = n`	i has n children
`#i:arity = n,m`	i has $n \leq k \leq m$ children
`#i:tokenarity = n`	i covers n tokens
`#i:tokenarity = n,m`	i covers $n \leq k \leq m$ tokens

- *Coverage* operations are used to express how two spans i and j overlap.
- *Precedence* operations express how many tokens two spans are apart in the primary text. For spans covering multiple tokens, the left-most covered token is used if the span is on the right-hand side of the operator and the right-most covered token is used if the span is on the left-hand side of the operator.
- *Dominance* operations relate two spans by their relative position in a dominance hierarchy.
- *Pointing relation* operations specify that two spans are linked by an edge with a given name.

Table 2 describes the syntax of every binary linguistic constraint including their variants. Both the pointing relation operator and the dominance operator evaluate a graph of named edges. The name is mandatory for the pointing relation operator because it is used to distinguish the type of relation expressed by an edge. For the dominance operator the name is optional. The normal usage omits the name and evaluates the entire dominance hierarchy. However, some corpus sources use different edge types to encode parts of the dominance hierarchy; see section 3.1 for details. The user can restrict the dominance operator to a particular type of dominance edge by using the named variant `>name` and `$name` instead of `>` and `$`.

2.3.5 Document meta data

By default, Annis will search every document in the corpus hierarchy below a root corpus selected by the user. It is possible to restrict the search to documents annotated with certain meta data, using the expression `meta::key="value"`. Similarly to node annotations, the value of a meta annotation can also be specified using a regular expression, or can be negated or omitted. Note that although the syntax is similar to node annotations, meta annotation definitions do not count as search terms and are skipped when evaluating search term references. They are also not considered when evaluating ORs. A document will only be searched if all meta annotations in the query are satisfied, regardless of the alternative in which they appear.

Table 2: Binary linguistic constraints in AQL.

Operator	**Definition**	
	Coverage operations	
`#i _=_ #j`	$i_{\text{left}} = j_{\text{left}} \wedge i_{\text{right}} = j_{\text{right}}$	Exact Cover
`#i _i_ #j`	$i_{\text{left}} \leq j_{\text{left}} \wedge i_{\text{right}} \geq j_{\text{right}}$	Inclusion
`#i _l_ #j`	$i_{\text{left}} = j_{\text{left}}$	Left Align
`#i _r_ #j`	$i_{\text{right}} = j_{\text{right}}$	Right Align
`#i _ol_ #j`	$i_{\text{left}} \leq j_{\text{left}} \leq i_{\text{right}} \leq j_{\text{right}}$	Left Overlap
`#i _or_ #j`	$j_{\text{left}} \leq i_{\text{left}} \leq j_{\text{right}} \leq i_{\text{right}}$	Right Overlap
`#i _o_ #j`	$i_{\text{left}} \leq j_{\text{right}} \wedge j_{\text{left}} \leq i_{\text{right}}$	Overlap
	Precedence operations	
`#i . #j`	i directly precedes j (same as `#i .1 #j`)	
`#i .* #j`	i indirectly precedes j	
`#i .n #j`	i precedes j with distance n	
`#i .n,m #j`	i precedes j with distance $n \leq k \leq m$	
	Dominance operations[a]	
`#i > #j`	i directly dominates j (same as `#i >1 #j`)	
`#i >[key="value"] #j`	i directly dominates j; the edge is annotated with the specified annotation[b]	
`#i >* #j`	i indirectly dominates j	
`#i >n #j`	i dominates j with distance n	
`#i >n,m #j`	i dominates j with distance or $n \leq k \leq m$	
`#i >@l #j`	j is the left-most child of i[c]	
`#i >@r #j`	j is the right-most child of i[c]	
`#i $ #j`	i and j are siblings	
`#i $[key="value"] #j`	i and j are siblings; both edges are annotated with the specified annotation	
`#i $* #j`	i and j share an common ancestor	
	Pointing relation operations[d]	
`#i ->name #j`	i directly points to j via a specifically named edge	
`#i ->name [key="value"] #j`	i directly points to j; the edge is annotated with the specified annotation	
`#i ->name * #j`	i points to j, either directly or through intermediate nodes; every edge along the path has the same name	
`#i ->name n #j`	i points to j with distance n	
`#i ->name n,m #j`	i points to j with distance $n \leq k \leq m$	

[a] There also exist named variants `>name` and `$name` for the unnamed dominance, sibling, and common ancestor operators.

[b] Edge annotations can be specified in the same way as node annotations, i.e., using regular expressions or by negating or omitting the value. It is also possible to specify multiple edge annotations by separating them with spaces.

[c] An edge annotation can also be specified using the variants `#i >@l [key="value"] #j` or `#i >@r [key="value"] #j`.

[d] The name is mandatory because it is used to specify the type of relationship modeled by the pointing relation edge.

3 Implementation of Annis on a relational database system

This section describes how Annis can be implemented on top of a relational database. We explain how an Annis annotation graph is stored as a set of relations and illustrate the translation of Annis queries to SQL. At the end of the section, we briefly discuss the current implementation on top of PostgreSQL. A more detailed description of Annis on PostgreSQL is available in [Ros11].

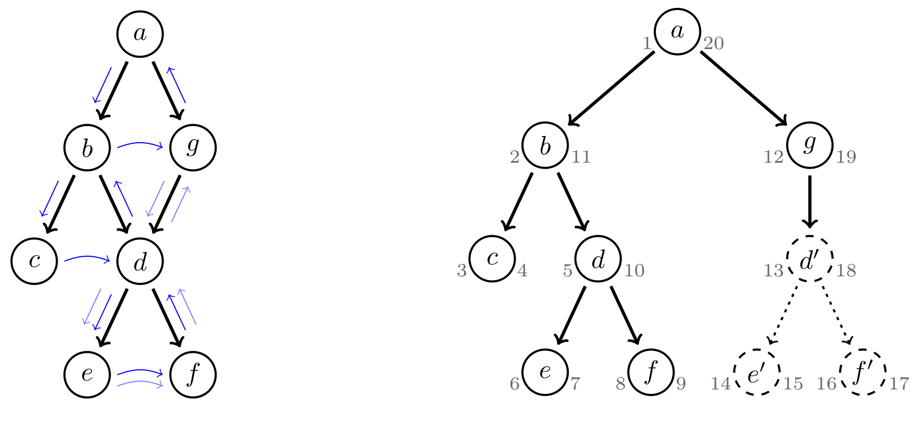
3.1 Storing the Annis model in a relational database

The annotated text spans of an annotation graph may be arranged in one or more hierarchies of arbitrary depth using different edge types. Before the graph is stored in a database, it is split into a set of connected components, so that each component contains only one type of edge. The structure of these components is encoded relationally using a combined pre/post-order scheme [GKT04].

3.1.1 The pre/post-order scheme

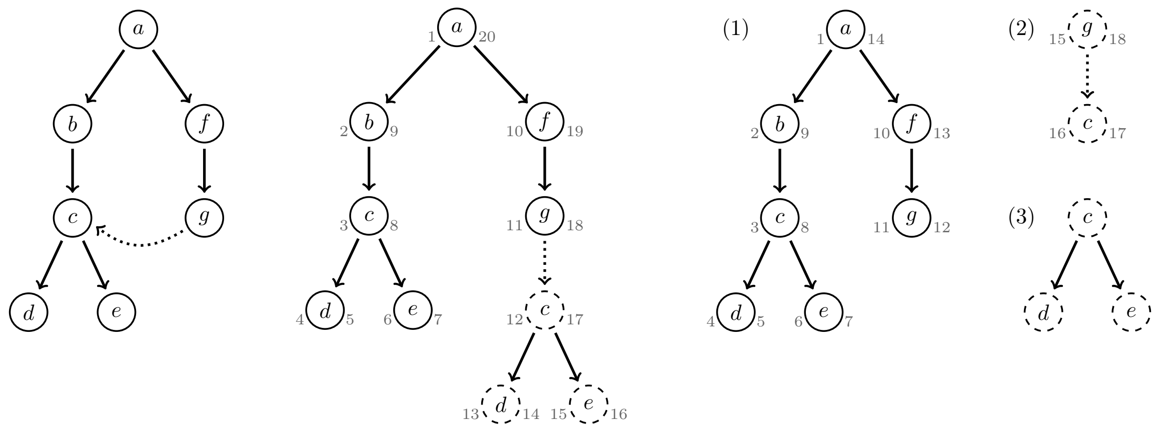
The pre/post order scheme encodes the structure of a DAG by assigning each node one or more pairs of pre-order and post-order ranks. Starting from a root node, the graph is traversed depth-first, indicated by dark blue arcs in Figure 8a. Each node is assigned a pre-order value when the traversal reaches the node before its children are visited and a post-order value after its children have been visited. One counter is used for both the pre-order and the post-order ranks. If the graph is multi-rooted, the traversal is repeated for each root node. The combined pre/post-order scheme allows an efficient reachability test: There exists a path between v and w if and only if $pre_v < pre_w < post_v$. Such a test is used to implement the indirect variants of the pointing relation and dominance operator.

In a DAG, a node may have more than one parent, as long as there are no directed cycles. Such nodes will be visited multiple times by the traversal. During each visit, a separate set of pre/post-order values is generate, all of which must be stored to encode the complete structure of the graph. In addition, the descendents of a node with many parents are also visited multiple times and accordingly also have multiple pre/post-order values assigned to them [TL05]. In Figure 8a, the nodes d, e and f are visited twice as indicated by the light blue arcs. In Figure 8b a tree is constructed from the pre/post-order values generated by the traversal. A node's pre-order value is displayed at its left side and its post-order value at its right side. The second set of pre/post-order values of d, e, and f is assigned to virtual



(a) Traversal (b) Decomposed tree

Figure 8: Assignment of pre/post-order values in a DAG. Adapted from [Vit04].



(a) Annotation graph (b) Decomposed tree (c) Separated components

Figure 9: Component separation by edge type.

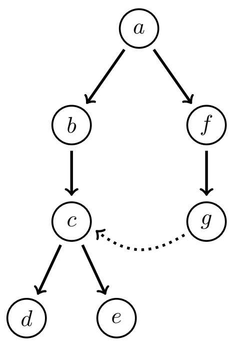
copies of these nodes. Evidently, the pre/post-order scheme introduces a certain amount of redundancy, e.g., the subgraph consisting of dotted edges in Figure 8b. This subgraph is already contained elsewhere in the graph. We will discuss the impact of these duplicate edges below.

3.1.2 Separation of connected components by edge type

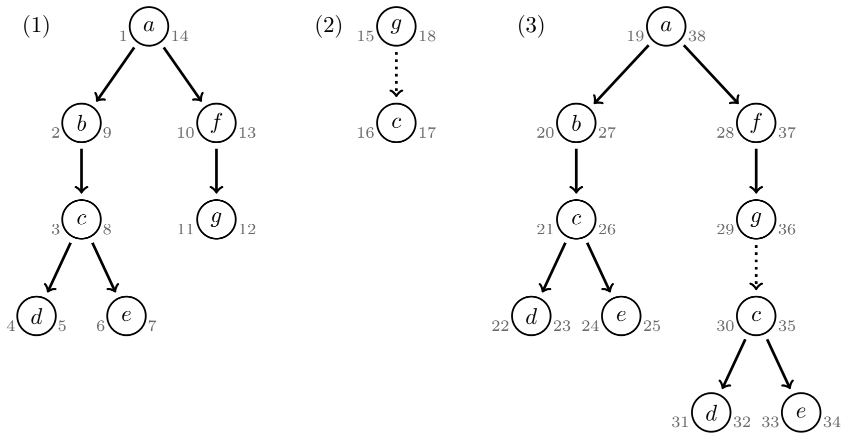
Queries on an hierarchical annotation type must only consider paths in which every edge is of the same type and has the same name. For example, in Figure 9a, solid edges denote a dominance hierarchy and the dotted edge a pointing relation. The node f does not dominate the node c because there exists no path from f to c consisting only of solid edges. However, if pre/post-order values are assigned to the original graph, as in Figure 9b, the condition $pre_f < pre_c < post_f$ is true, suggesting the existence of such a path. Since we cannot test the type of every edge along a path, we have to split the graph into connected components, shown in Figure 9c, so that each component only contains edges of one type. This process generates multiple pre/post-order values for nodes which are leaves in one component and the root of another component of a different type, such as the node g in Figure 9c. Conversely, it may separate multiple incoming edges of nodes such as c, making the annotation graph more tree-like and reducing the inherent redundancy of the pre/post-order scheme. For example, in Figure 9b the maximum post-order rank is 20, i.e., there are ten pre/post-order pairs. In Figure 9c there are only nine pre/post-order pairs assigned to the nodes in the components (1) and (2). No pre/post-order values have to be generated for the nodes in component (3) because it is already contained in component (1).

3.1.3 Merging of dominance hierarchies

In some corpora, e.g., the TIGER Treebank, there are two separate types of dominance edges. Normal edges make up the dominance hierarchy, while secondary edges link discontinuous constituents in coordinated phrases. An example of a syntax tree containing a secondary edge is shown in Figure 16 on page 37. TIGERSearch does not take secondary edges into account when evaluating a dominance query. Instead, the user has to explicitly ask for these edges using a separate language construct. In Annis, we would like to consider paths containing both types of edges for the dominance operator while still keeping the distinction between the two edge types if the user queries it explicitly. For example, Figure 10a shows a syntax graph with a dotted secondary dominance edge. If the graph is separated by edge type, e.g., components (1) and (2) in Figure 10b, the path from f to c is lost. These two components can be used to answer dominance queries which specify the type of dominance edge. We also need to combine



(a) Dominance graph



(b) Decomposed dominance trees

Figure 10: Merging of dominance hierarchies.

the separate dominance hierarchies into the merged component (3) containing both types of dominance edges in order to answer dominance queries without further specifying the dominance type. Note that the encoding of the merged component may require more pre/post-order values than the individual components containing only one edge type taken together. For example, the maximum post-order rank of components (1) and (2) is 18, i.e., there are nine pre/post-order pairs, whereas in component (3) there are ten pre/post-order pairs.

3.1.4 Redundancy in the pre/post-order scheme

As we have already discussed, the encoding of the annotation graph using pre/post-order values may introduce some redundancy, e.g., there may be more pairs of pre/post-order values than there are edges in the graph. Indeed, every time an already encountered node v is visited again during the traversal, a new pair of pre/post-order values is generated for each node in the subgraph below v. The amount of duplicate pre/post-order pairs depends on the number of non-tree edges in the DAG as well as on their location. A non-tree edge that is closer to the leaves of the DAG will generate fewer duplicates than a non-tree edge that is close to the root of the DAG.

In practice, the largest amount of duplication is not generated by visiting nodes multiple times, but because of the merging of separate dominance hierarchies. As a result, each connected component is stored once individually and a second time as part of the merged component. This replication at least doubles the number of pre/post-order values, as Figure 10b shows. The edge type of the merged component is set to a value that is distinct from other dominance edge types. Thus, the set of pre/post-order values is partitioned by the edge type into distinct subsets and we only have to consider one of those subsets during the evaluation of a dominance operation. The redundancy of pre/post-order values for a node in a particular subset is unaffected by the node's pre/post-order values in other subsets. We can therefore examine the redundancy in each subset individually. For example, the nodes c, d, and e in component (1) in Figure 10b have only one pair of pre/post-order values each, even though they have two pairs in component (3).

The separation of the annotation graph into connected components results in the generation of additional pre/post-order values due to the creation of new roots in the individual components. However, the individual components are more tree-like and structures that previously required multiple pre/post-order values, e.g., component (3) in Figure 9c, can be pruned. In practice, the remaining individual components contain very few non-tree edges and therefore the redundancy introduced by the pre/post-order scheme is quite low. For example, in the merged dominance hierarchy of the TIGER corpus, 0.6%

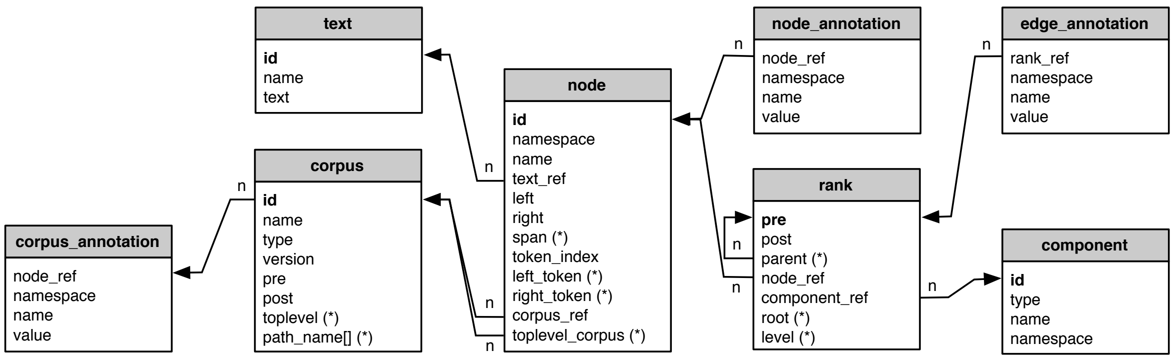


Figure 11: Annis database schema.

of edges are non-tree edges. Consequently, there are only 1.3% more pre/post-order values encoding the merged dominance hierarchy than there are dominance edges.

Given this low amount of overhead, the pre/post-order scheme presents a good trade-off between the efficiency of answering reachability queries in a DAG and the space requirements of the indexing scheme [TL05]. Two popular alternative approaches are recursive queries and the computation and storage of the transitive closure of the DAG. A recursive query needs no additional information, but the time required for its evaluation is dependent on the length of the path and generally prohibitive. The transitive closure as an index is usually faster, but requires considerably more resources to store.

3.1.5 Database schema

Figure 11 depicts the database schema that is used to store an Annis annotation graph in a relational database. The attributes marked by an asterisk contain data that is not strictly necessary to encode the annotation graph. They can be derived from the unmarked attributes, but are stored in the database to speed up the evaluation of certain Annis operators. Some of these values are computed before the corpus is imported and are already contained in the relANNIS files, while others are computed during the corpus importation process.

Each entity in the graph, i.e., nodes, edges, and annotations, has a name. It is used as a database-independent identifier of nodes, to encode the subtype of an edge, and as the key of an annotation key-value pair respectively. The name of a node is typically derived from an XML `id` attribute, but is not guaranteed to be unique. Names are optionally prefixed with a namespace which may be used to group data belonging to the same annotation layer. By convention it is set to the name of the tool that was used to create the annotation data.

The `text`, `node`, and `node_annotation` tables

A tuple in the `node` table refers to the text span from the *character* at index `left` to `right` (inclusive) of the text referenced by the foreign key `text_ref`. If the text span is a token, the `span` attribute contains the actual content of the token and the `token_index` attribute its position in the text. For non-tokens, both attributes are set to **`NULL`**. The attributes `left_token` and `right_token` contain the index of the left-most and right-most *token* covered by the current text span respectively. For tokens, these attributes are set to the value of `token_index`. The attributes `left_token` and `right_token` are used to implement precedence operators independently of coverage operators which use the attributes `left` and `right`. The `corpus_ref` attribute refers to the document containing the span and the `toplevel_corpus` attribute to the document's root corpus.

Node annotations are stored in the node_annotation table. The annotation name is used as the annotation key.

The text table stores the full contents of a primary text, i.e., the text of the corpus without its annotations, in the text attribute. It is used to construct the textual contents of non-token text spans during the evaluation of $MATRIX$ queries.

The component, rank, and edge_annotation tables

The component table contains an entry for every connected component in the annotation graph, partitioned by edge type and name. Since every edge in a component has the same type and name, this information is also stored in the component table.

A tuple in the rank table stores a pair of pre/post-order values of the node referred to by the node_ref foreign key. The parent attribute is a foreign-key reference to the pre attribute of the parent node and the component_ref attribute is a foreign-key reference to a component containing the node. A row in the rank table in which the parent attribute is not **NULL** can be interpreted as an *incoming* edge of the node specified by the node_ref foreign key with the type and name specified by the component_ref foreign key. If parent is **NULL**, the row refers to a root node in the component referenced by the component_ref. The rank table also contains a root attribute that is set to **TRUE** if a node is a true root in the annotation graph, i.e., the node is a root in all components containing the node. Finally, the level attribute stores the depth of the node in the component referenced by component_ref.

Edge annotations are stored in the edge_annotation table similarly to node annotations.

The corpus and corpus_annotation tables

The corpus table contains an entry for each root corpus and its subcorpora or documents. The corpus hierarchy is encoded using a combined pre/post-order scheme. Since a document may only have one parent document, a separate table to store the pre and post-order ranks is not needed. The attributes name, type, and version are currently not used by the Annis service. The boolean attribute toplevel signals a root corpus. Finally, the path_name array attribute enumerates the names of every document along the path from the root corpus to the document encoded by a corpus tuple. This information is also used by the web interface to visualize the context of matches.

Corpus annotations are stored in the corpus_annotation table similarly to node and edge annotations.

The node table contains two foreign-key references to the corpus table. The toplevel_corpus attribute is used to restrict the search to root corpora selected by the user in the web interface. The corpus_ref attribute points to the document containing the span encoded by the node tuple. It is used to restrict the search to documents matching a set of corpus annotations.

3.2 Computing the solutions to an Annis query

The first step of evaluating an Annis query is the computation of its solutions, i.e., the subgraphs of the annotation graph matching the template defined by the query. Based on these solutions, query functions then retrieve additional data from the corpus. A solution for a query with n search terms is represented by an n-tuple where the i-th element is the primary key of the node tuple representing the text span matched by the i-th search term. The SQL query that computes the solutions to an Annis query with n search terms follows the template shown in Listing 2.

3.2.1 Tables required for the evaluation of search terms

For each search term of the Annis query, the **FROM** clause of the SQL query shown in Listing 2 will list an alias of the node table because a text span matched by a search term is identified with the primary

```
                        n times
SELECT DISTINCT node1.id AS id1, ..., nodeN.id AS idN
FROM   node AS node1 JOIN additional tables required to evaluate the first search term,
       ...,
       node AS nodeN JOIN additional tables required to evaluate the n-th search term
WHERE  predicates on attributes of the selected tables to evaluate the query
```

Listing 2: SQL query template to compute the solutions of an Annis query.

key of the tuple representing the text span in `node`. Depending on the type of search term and how it is used in linguistic constraints, additional table aliases may have to be joined: Annotation search terms require the `node_annotation` table for evaluation. The `rank` and `component` tables have to be joined if the search term is used in a dominance or pointing relation operation. The `rank` table is also required if the search term is qualified with the **`root`** operator. Finally, if the search term is used as the right-hand-side argument of an edge-annotated dominance or pointing relation operation the `edge_annotation` table has to be joined. One `edge_annotation` table alias is required for each edge annotation specified in the operation.

3.2.2 Annis queries containing OR

An Annis query q containing OR is evaluated as follows: First, the query is transformed into its disjunctive normal form $q' = \bigvee_{i=1}^{k} q_i$ with k alternatives. Next, the maximum number of search terms m in any alternative is determined. Then, for each alternative q_i an SQL query fragment is generated according to the template in Listing 2. If the alternative has n search terms and $n < m$, the **`SELECT`** clause is padded with $m - n$ **`NULL`** terms. Finally, the SQL fragments are concatenated into one SQL query with **`UNION`**. As an example of this process, Listing 3 depicts an SQL query template for an Annis query with two alternatives q_1 and q_2. q_1 has n search terms, q_2 has m search terms, and $n < m$.

```
                              n times               m − n times
SELECT DISTINCT node1.id AS id1, ..., nodeN.id AS idN, NULL, ..., NULL
FROM   node AS node1 JOIN ...,
       ...,
       node AS nodeN JOIN ...
WHERE  predicates to evaluate q1
                                  m times
UNION SELECT DISTINCT node1.id AS id1, ..., nodeM.id AS idM
FROM   node AS node1 JOIN ...,
       ...,
       node AS nodeM JOIN ...
WHERE  predicates to evaluate q2
```

Listing 3: SQL query template for an Annis query with multiple alternatives.

3.3 Implementation of select Annis language features

Many Annis language features evaluate one or more text span properties for which there exists a corresponding table attribute in the SQL schema. Binary linguistic operations compare properties of two spans which have been declared previously in the query. The implementation of these features is fairly straightforward. The main exception is the common-ancestor operator which tests whether two spans are connected by a third, implicit parent span. The implementation of search terms containing a regular expression is also more complicated because regular expression operations differ greatly in various RDBMS. Below, we illustrate the SQL implementation of Annis language features by means of a few select operations.

3.3.1 Implementation of search terms

A text search is implemented by testing the content of the `node.span` attribute. The generic token search **`tok`** can be implemented in two ways: By a **`NOT NULL`** predicate on `node.span` or a **`NOT NULL`** predicate on `node.token_index`. An annotation search is implemented by joining the `node_annotation` table and testing the content of `node_annotation.name` and, if necessary, of `node_annotation.value`. The generic **`node`** search term requires no predicates in the **`WHERE`** clause. Simply listing a `node` table alias in the **`FROM`** clause and selecting the primary key `node.id` in the **`SELECT`** clause is sufficient to implement this search term.

3.3.2 Implementation of coverage operators

Coverage operations compare the left and right text borders of two spans taken from the same text. The left and right borders of a span are modeled by the attributes `node.left` and `node.right`. A reference to the primary text from which the span is taken is stored in the attribute `node.text_ref`. To implement coverage operations in SQL, we only have to substitute the corresponding table attribute for the text span attributes referenced in the comparisons in Table 2. For example, the Exact Cover operation `#1 _=_ #2` is implemented by the following SQL predicates:

```
node1.text_ref = node2.text_ref AND
node1.left = node2.left AND
node1.right = node2.right
```

3.3.3 Implementation of precedence operators

In Annis, precedence within a text is defined in terms of an explicit order on the tokens of a text. This order is determined by a property modeled by the attribute `node.token_index`. It can be extended to non-token spans s and t by comparing the index max_s of the right-most token covered by s with the index min_t of the left-most token covered by t, i.e., $s <_{pos} t := max_s < min_t$. Using the order relation $<_{pos}$ we can formally define the precedence operations as follows:

$$\begin{array}{lcl} \texttt{\#i .* \#j} & \Longleftrightarrow & max_i < min_j \\ \texttt{\#i . \#j} & \Longleftrightarrow & max_i = min_j - 1 \\ \texttt{\#i .n \#j} & \Longleftrightarrow & max_i = min_j - n \\ \texttt{\#i .n,m \#j} & \Longleftrightarrow & min_j - m \leq max_i \leq min_j - n \end{array} \tag{1}$$

For each span, the index of the left-most and right-most tokens covered by the span are stored in the attributes `node.left_token` and `node.right_token`. To implement precedence operations in SQL, we substitute the corresponding table attribute for the text span attributes referenced in the comparisons in Equation 1. For example, the direct precedence operation `#1 . #2` is implemented by the following SQL predicates:

```
node1.text_ref = node2.text_ref AND
node1.right_token = node2.left_token - 1
```

3.3.4 Implementation of dominance and pointing relation operators

As described in section 3.1, the annotation graph of a corpus is stored in an RDBMS using a pre/post-order scheme. Additionally, operators that evaluate the position of two spans in the annotation graph require that the nodes representing the spans are contained within a single, typed and named component. The pre and post-order ranks of a node are stored in the attributes `rank.pre` and `rank.post`, while the component type and name are stored in the attributes `component.type` and `component.name`.

To implement a dominance and pointing relation operation in SQL, we have to join both tables to the span's `node` table alias, compare the pre and post values of both spans, and make sure that both nodes are contained in an appropriate component. For example, the indirect dominance operation `#1 >* #2` can be implemented by the following SQL predicates:

```
1 rank1.pre < rank2.pre AND
2 rank2.pre < rank1.post AND
3 component1.id = component2.id AND
4 component1.type = 'd' AND
5 component1.name IS NULL
6 component2.type = 'd' AND
7 component2.name IS NULL
```

Strictly speaking, the predicate in line 3 is not required because the pre and post-order ranks are computed in such a way that the comparison in lines 1 and 2 holds if and only if both nodes are in the same component. However, making this relationship explicit in the SQL query provides additional information for the query optimizer to exploit. Lines 4 and 6 test if the component is a dominance component; for pointing relation components the type is `p` instead of `d`. Lines 5 and 7 restrict the test to the merged dominance hierarchy. The component tests are attached to the both sides of the operator even though the component is the same in both cases. In general, we provide as much information as possible in the SQL query for the benefit of the query optimizer.

The direct variants of the dominance and pointing relation can test the `parent` attribute instead of a potentially costly range predicate on the pre and post-order ranks. The direct variants also allow the specification of one or more edge annotations. To implement an edge-annotated operation the table `edge_annotation` has to be joined to the `node` table alias representing the span on the right-hand side of the operator because an entry in `rank` is interpreted as an *incoming* edge of the span specified by `rank.node_ref`. For example, the edge-annotated direct dominance operation `#1 >[func="OA"] #2` can be implemented by the following SQL predicates:

```
rank1.pre = rank2.parent AND
component1.id = component2.id AND
component1.type = 'd' AND
component1.name IS NULL AND
component2.type = 'd' AND
component2.name IS NULL AND
edge_annotation2.name = 'func' AND
edge_annotation2.value = 'OA'
```

The sibling operator can be implemented by testing the `rank.parent` attribute of both spans. If an edge annotation is specified the `edge_annotation` table needs to be joined to both spans and tested.

3.3.5 Implementation of the common ancestor operator

The common ancestor operator does not fit the general template for dominance operations shown above. It requires a search for a span that is an ancestor of both source spans in the same component. This is implemented as a nested subquery in an **EXISTS** clause as shown below for the operation `#1 $* #2`. The subquery is correlated which is usually an indicator of bad performance. However, it is guarded both inside and outside by a test on the `component.id` attribute. As a consequence of the outside test in line 1, the evaluation of the **`EXISTS`** clause can be skipped in many cases. The inside test in line 7 restricts the search space considerably, to about eleven nodes in a component on average in the TIGER Treebank (see Table 9 on page 43). Furthermore, the evaluation of the **`EXISTS`** clause can finish as soon as one common ancestor is found and does not have to enumerate them (line 6).

```
1  component1.id = component2.id AND
2  component1.type = 'd' AND
3  component1.name IS NULL AND
4  component2.type = 'd' AND
5  component2.name IS NULL AND
6  EXISTS (SELECT 1 FROM rank AS ancestor WHERE
7    ancestor.component_ref = rank1.component_ref AND
8    ancestor.pre < rank1.pre AND rank1.pre < ancestor.post AND
9    ancestor.pre < rank2.pre AND rank2.pre < ancestor.post)
```

3.3.6 The *COUNT* query function

The *COUNT* query function returns the number of distinct solutions to an Annis query. For queries consisting of only one alternative and n search terms this can be achieved by modifying the **`SELECT`** clause as follows: **`SELECT count(DISTINCT`** `node1.id,..., node`N`.id)`. However, if the query consists of multiple alternatives joined by **`UNION`**, this strategy returns the counts for each alternative separately. They would then have to be added in a second step. Instead, we have implemented the *COUNT* query function as shown in Listing 4: The SQL query computing the solutions to the original Annis query is wrapped as a nested subquery in the **`FROM`** clause between lines 2 and 4. The number of solutions is counted in the outer query in line 1. It is not necessary to include the **`DISTINCT`** operator in the outer query because the query solutions have already been deduplicated by the nested subquery. A similar strategy is used to implement the *ANNOTATE* and *MATRIX* query functions which are described in more detail in appendix A.

```
1  SELECT count(*)
2  FROM   (
3                SQL subquery to compute query solutions as described in Listing 2 and Listing 3
4         ) AS solutions
```

Listing 4: SQL query template for the *COUNT* query function.

3.3.7 Regular expression searches

In Annis, text and annotation searches can be formulated with the help of a regular expression, following a POSIX or Perl-compatible syntax [IEE04]. The SQL:2003 standard [ISO03] defines two different and incompatible ways to evaluate regular expressions: First, the **`LIKE_REGEX`** predicate (Feature F841) matches a string against an XQuery regular expression. XQuery regular expressions implement a superset of the POSIX regular expression syntax and expressiveness [MMWK10]. Second, the **`SIMILAR TO`** predicate (Feature T141) matches a string against a SQL regular expression. SQL regular expressions differ from POSIX regular expressions in their treatment of the `.` (dot) and the _ (underscore) character. In POSIX regular expressions, the dot is a placeholder for any character whereas the underscore is interpreted literally. In SQL regular expressions the situation is reversed: The dot is interpreted literally and the underscore serves as a placeholder.

Unfortunately, neither predicate is particularly useful to implement regular expression text and annotation searches. The **`SIMILAR TO`** predicate requires a translation from POSIX-style regular expressions to

SQL regular expressions. This process may introduce subtle errors or irregularities in the evaluation of these searches which may not be immediately apparent to the user. As far as we know, the **`LIKE_REGEX`** predicate is not supported by any major commercial or open-source RDBMS. Thus, the implementation of regular expression searches is specific to the underlying RDBMS. In PostgreSQL, we use the ~ (tilde) operator which matches the string on its left-hand side to the regular expression on its right-hand side.

3.3.8 Corpus selection

Typically, a user selects a list of corpora in the web front-end on which to evaluate an Annis query. He can further restrict the evaluation to those documents which match meta data specified in the query. For the remainder of this work we assume that the database contains a single corpus and that the query does not contain any meta annotations. In appendix A, we describe the necessary steps to restrict a query to a set of corpora or annotated documents in more detail.

3.4 Current implementation on top of PostgreSQL

It is not efficient to evaluate Annis queries on a large corpus that is stored using the schema consisting of normalized tables as described in section 3.1.5, also called the *source schema*. For example, the evaluation of the query shown in Listing 1 on page 17 on the TIGER Treebank[5] on a decent server[6] requires at least 2.2 seconds. This duration does not seem overly long. However, the query is not very complex and even such a small delay has an impact on the attitudes and performance of users in interactive computer tasks [GHMP04]. Even worse, on a consumer laptop the query will not even finish within 60 seconds. As one goal of Annis is to make large corpora accessible to researchers working on their own hardware, such a performance is unacceptable.

One reason for the slow performance of the source schema is the joining of multiple tables to construct a span that is referred to by a search term in an Annis query, particularly if that span is referenced by a dominance or pointing relation operation. Figure 12 shows the query execution plan generated by PostgreSQL to evaluate the Annis query shown in Listing 1 on page 17 on the TIGER Treebank using the source schema. A total of 12 tables, four for each search term, have to be joined. The shaded areas

[5]The TIGER Treebank is described in detail in section 5.1.
[6]The test systems are described in detail in section 5.3.

Table 3: Evaluation time (in ms) of some Annis queries on the TIGER Treebank.

Query	Server		Laptop	
	Source schema	Materialized schema	Source schema	Materialized schema
`node`	**876**	2888	**1199**	3810
`tok`	**733**	3120	**958**	3863
`cat="S" & pos="VVFIN" & #1 _i_ #2`	**1980**	4860	**2394**	6600
`cat="S" & pos="VVFIN" & #1 > #2`	4904	**369**	18080	**776**
`cat="S" & cat="NP" & #1 >[func="OA"] #2 & cat="NP" & #1 >[func="SB"] #3 & #2 .* #3`	*> 60 s*	**276**	*> 60 s*	**599**
`cat="S" & #1:root & pos="VVFIN" & #1 > #2 & node & #1 >[func="SB"] #3 & #2 .* #3`	3669	**535**	*> 60 s*	**7336**

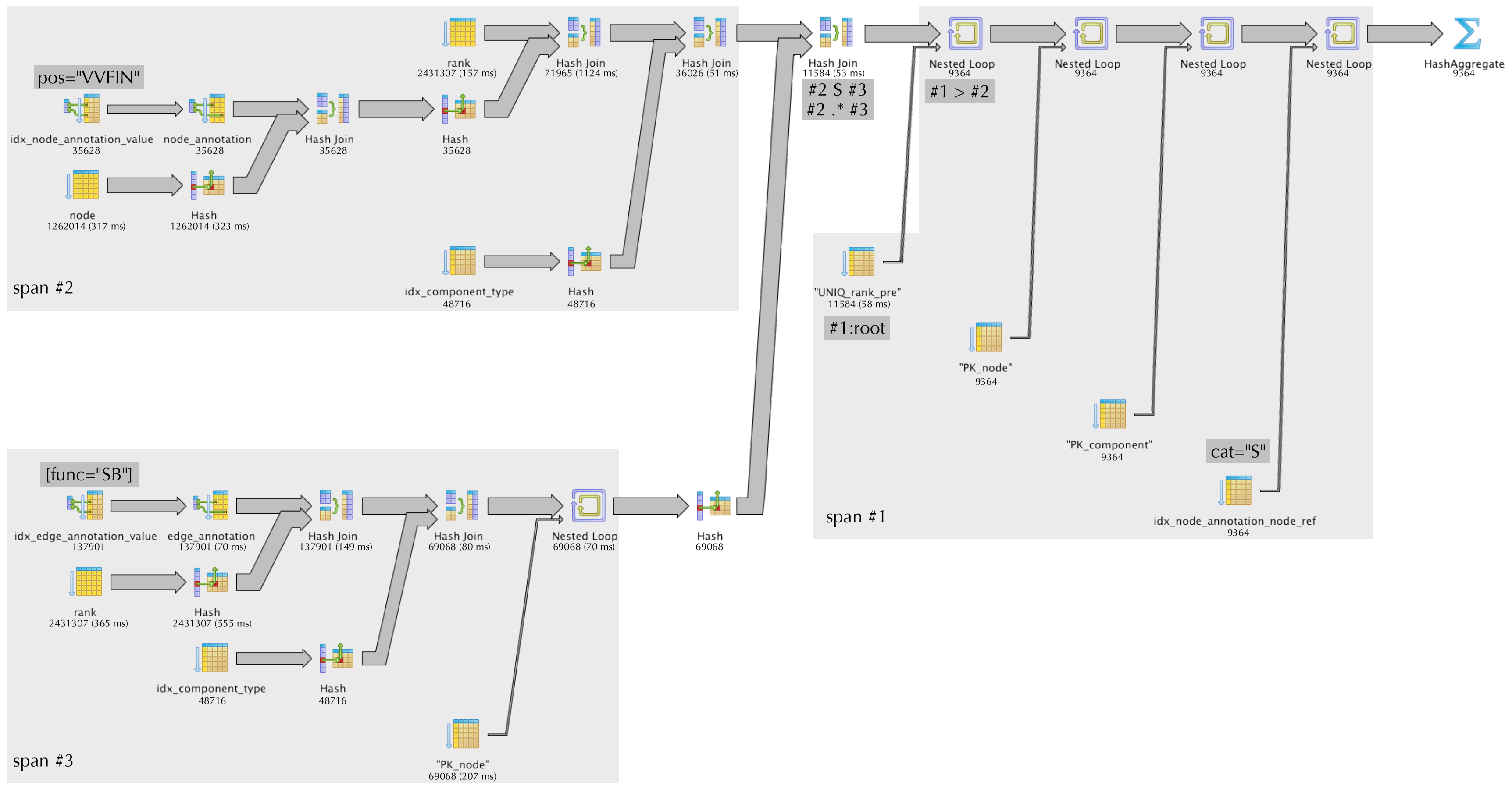
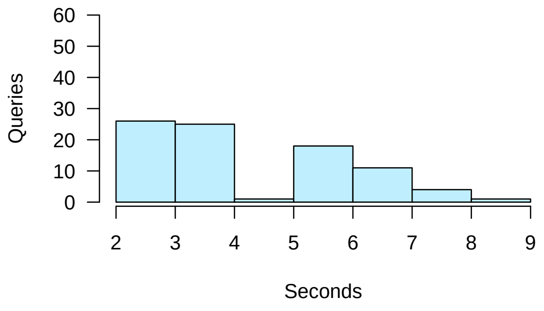
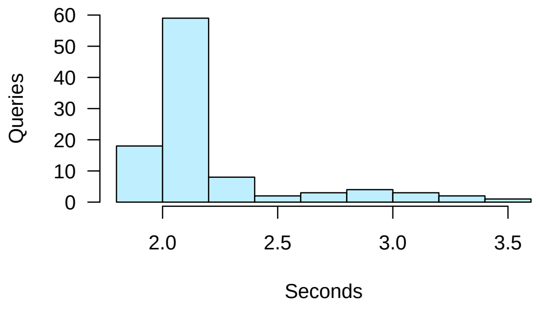


Figure 12: Query execution plan for the Annis query shown in Listing 1 on page 17 evaluated on the source schema.



(a) Histogram size: 100 entries (14 timeouts)



(b) Histogram size: 10 entries

Figure 13: Frequency distribution of query runtime depending on statistics.

combine the table and/or index scans as well as join operations that are required to construct a span referenced by a single search term in the query. Printed below each operation is the number of returned rows or the number of scans for the inner table of a nested loop join. Additionally, the execution time of the operation is listed if it requires more than 100 ms. Operations implementing a specific part of the Annis query are labeled accordingly. Note that the dominance operation `#1 > #3` is rewritten by the PostgreSQL query execution engine to the sibling operation `#2 $ #3` which is equivalent for this query. Table scans and joins implementing a specific part of the Annis query are distributed across the entire query plan leading to large intermediate results that are discarded later on. This query plan requires 3.7 seconds to evaluate.

Additionally, the evaluation on the source schema is nondeterministic, insofar that the chosen query execution plan depends on the statistics kept by PostgreSQL. For each attribute in a database schema, PostgreSQL stores the frequency of the most common values in its catalog. This histogram is used by the query optimizer in its estimation of the costs of a particular query. The size of the histogram can be set for each attribute individually, ranging from one to 10000 entries, with the default size of 100 entries. Unless the table is very small, PostgreSQL generates the histogram by scanning a random sample of the table, i.e., its composition will be different every time it is regenerated. As Figure 13a shows, the contents of the histogram can have a profound effect on the runtime of a query. The chart depicts the frequency distribution of the results of 100 experiments running the query shown in Listing 1 on the TIGER Treebank using the normalized schema. For each experiment, the statistics of the `node_annotation` attribute were regenerated using the default size of 100 entries, and the best time of five consecutive runs was determined. The effect of the random sample is a large spread between 2 and 9 seconds required to evaluate the query if the default histogram size of 100 entries is used. In 14 cases, PostgreSQL created a query plan that did not finish within 60 seconds. Contrary to intuition, as Figure 13b shows, the behavior is improved by *reducing* the amount of statistics kept by PostgreSQL. However, we do not know if this result can be generalized to other queries.

Performance can be improved by materializing the result of common joins. It is not possible to construct a materialized view specifically for every query because the number of accessed tables depends on the number and type of search terms in a query and on the linguistic constraints referencing the search terms. However, the information about a single span that is distributed across the five tables `node`, `rank`, `component`, `node_annotation`, and `edge_annotation` in the source schema can be collected in a `facts` table. This is called the *materialized schema*. The `facts` table is accessed once for each search term and is joined with itself to implement binary linguistic constraints. This is illustrated by the query plan in Figure 14. Since the `facts` table contains all the information about a span it is possible to construct dedicated indexes which can be used to implement a search term selection and a binary linguistic constraint in one index scan. For example, the partial index `idx_1971_nav_pos_dom_parent` in Figure 14 only indexes those rows in `facts` for which the conditions `node_annotation_name = 'pos'` and `edge_type = 'd'` are true. The indexed columns are `node_annotation_value` and `parent`. This index is used to implement the search term `pos="VVFIN"` and returns the tuples in such an order that the dominance operation `#1 > #2` can be implemented efficiently by a nested loop or merge join. Rows encoding a different node annotation

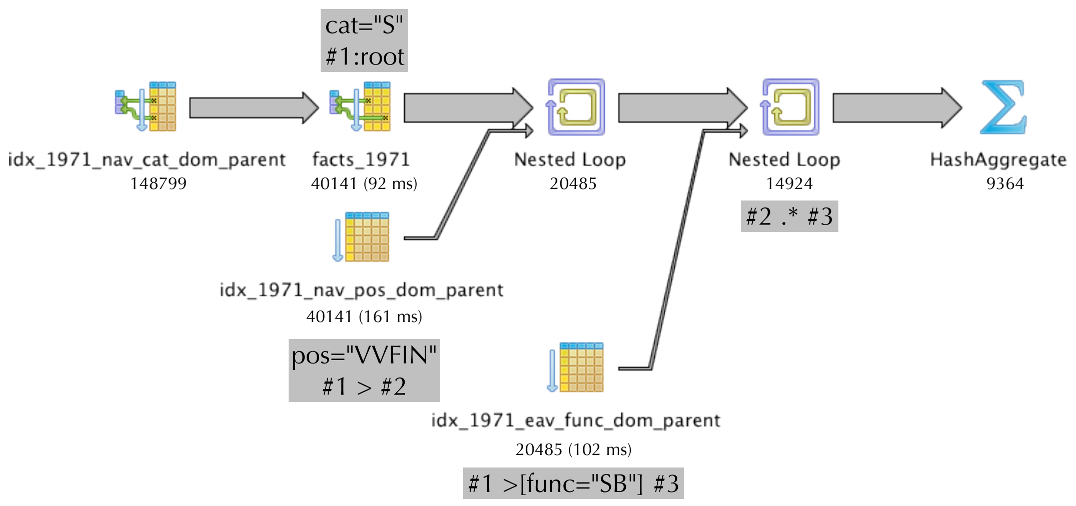


Figure 14: Query execution plan for the Annis query shown in Listing 1 on page 17 evaluated on the materialized schema.

name or another edge type are skipped. The indexing scheme is described in more detail in [Ros11]

The materialized schema has disadvantages. The first is increased disk space consumption. As Table 4 shows, the size of the materialized `facts` table is more than twice the size of the individual tables of the source schema. The additional indexes further increase the required disk space by a factor of four. Furthermore, the performance of Annis queries is not improved uniformly as Table 3 shows. Whereas Annis queries containing dominance and pointing relation operations are considerably sped up, simple but often-used queries such as **`node`** or **`tok`** or queries which only use coverage or precedence operations are slowed down. Evaluations of regular expressions such as `/[Dd]as/`, in which the first character is not fixed, also require more time. The `facts` table acts as a materialized view of a common subexpression used in Annis queries, but it is not used transparently by the PostgreSQL query optimizer [CKPS95]. The Annis SQL compiler must decide whether to use the source schema, the materialized schema, or a combination thereof to answer an Annis query. Currently, the materialized schema is always used.

A second disadvantage of the materialized `facts` table is that it limits the flexibility of the database schema. For example, suppose that we wanted to model multiple token orders. The texts in a learner corpus are often annotated with insertions, deletions, or movement, in the case that the original author did not produce a grammatically correct sentence. There are effectively two token orders in such a corpus: The order used by the author and the order of the text that has been corrected by the annotator. This could be modeled by extracting the attributes `left_token` and `right_token` from the `node` table into a separate `precedence` table and adding a discriminating attribute similar to `component.type` for edges. Queries evaluated on the source schema that do not use a precedence operation should be unaffected by this change. However, if the join of the source tables including the new `precedence` table are materialized, the number of additional token orders increases the size of the `facts` table. This change would affect queries that do not use a precedence operation similarly to how the queries in Table 3 that do not use a dominance or pointing relation are affected by the evaluation on the materialized `facts` table.

Table 4: Disk usage (in MB) of the TIGER Treebank in PostgreSQL.

	Tables	**Indexes**	**Total**
Source schema	525	1407	1932
Materialized schema	1201	6707	7908

4 Related Work

In this section we survey other linguistic query languages. We first list a set of requirements that should be fulfilled by a linguistic query language. We then discuss XML-based and dedicated linguistic tree query languages as well as multi-layer query languages, and compare them to Annis.

4.1 Requirements for a modern linguistic query language

There exists a multitude of linguistic query languages, often developed for a specific corpus model or annotation scheme. Lai and Bird surveyed six languages for querying treebanks and tried to extract common features in an effort to devise a reusable linguistic query language [LB04]. In [LB10] they list requirements for a linguistic tree query language:

Linear order. At the most basic level a text is arranged sequentially. Users want to navigate this order, e.g., to find the text preceding the finite verb in a sentence. Note that this does not necessarily exclude overlap, e.g., in a conversational corpus when two persons speak at the same time.

Hierarchical order. Many linguistic annotation schemes are implemented as hierarchies of text spans, e.g., syntax or dependency trees. Other schemes link text spans without imposing a hierarchy, e.g., coreference annotations.

Interactions between linear order and hierarchy. Users want to query both the linear and hierarchical dimension simultaneously, e.g., find the left-most or right-most descendents of a non-terminal node in a syntax tree.

Boolean operations. Starting from simple queries users want to construct complex queries using conjunction, disjunction, and negation. Note that proper support of negation implies universal quantification.

Closures. Specifically users want to be able to impose restrictions on the steps involved in the closures of basic relations such as dominance and precedence. As an example, consider the sentence in Figure 17. It contains three levels of alternating, nested prepositional phrases (PP) and noun phrases (NP). This nested structure could in principle be extended to an arbitrary depth and users may want to search for such alternating structures.

Based on our experience with Annis, we can formulate additional requirements for a query language for multi-layer corpora, that go beyond those of tree query languages [ZRLC09]:

Arbitrary annotation layers. Many multi-layer corpora are built by extending existing syntactically annotated corpora with additional annotations covering a wide variety of linguistic fields. Thus, a multi-layer corpus query language should not impose any restrictions on the annotation layers contained in a corpus.

Overlapping annotations. Tree query languages are specifically designed to query syntax trees, i.e., a well-formed nested structure. In contrast, the annotations in a multi-layer corpus may overlap arbitrarily and the query language must provide the means to query these relationships.

Multiple, semantically different hierarchies. Syntax trees are only one example of a linguistic annotation that is modeled as a hierarchy. Multi-layer corpora can contain many concurrent hierarchical annotations over the same text. The query language must be able to distinguish between them and should allow the user to query them simultaneously.

Operations over one or more texts. In most tree query languages, the precedence operation is restricted to individual parse trees. Consequently, it is impossible to construct a query that crosses sentence boundaries. In a multi-layer corpus, users want to express relationships that may cover the entire text or even multiple texts in the case of parallel corpora.

4.2 XML-based query languages

With the widespread adoption of XML [BPSM+00] in the linguistic community, there have been many attempts to use XML-related languages to query linguistic corpora. For example, Bouma and Kloosterman used XPath [CD+99] to query the Alpino Dependency Treebank using an XML storage scheme that mirrored the dependency hierarchy with the XML tree structure and encoded the linear order of the text in XML attributes [BK02]. Taylor investigated the translation of dedicated linguistic query languages to XSLT [C+99] in order to simplify their implementation by reusing established technologies [Tay03]. Somewhat related, Schäfer used XSLT to connect different components in a NLP pipeline used to construct a syntax parser [Sch03]. Finally, XQuery [BCF+10] has been studied extensively as a linguistic query language, e.g., in [Cas02, RECD08, BK07], and [ABdVB06]. The latter uses MonetDB/XQuery [BGvK+06] which stores the XML graph in a RDBMS using an encoding based on pre/post-order traversal ranks.

Being Turing-complete, XSLT and XQuery clearly make it possible to formulate any kind of practical linguistic query [Kep04]. However, this typically presupposes at least some knowledge of the underlying data model and how it is encoded in XML. XSLT and XQuery also require at least some programming knowledge and may thus be inaccessible to corpus linguists if used to query a corpus directly. These disadvantages can be addressed if XSLT and XQuery are used as an intermediate layer of the querying interface, e.g., in [Tay03] or [RECD08].

Bird et al. and Cassidy argue that while XPath and XQuery are a good match for querying hierarchical constraints, their support for sequential constraints is weak [BCD+05, Cas02]. Bouma and Kloosterman disagree and propose to encode the linear order of the text as XML attributes independently of the hierarchical order [BK02]. They show that this scheme allows for a compact treebank encoding and for the concise formulation of many linguistic queries using XPath alone. For more complex relation extraction tasks they propose custom functions based on XQuery [BK07].

To address perceived shortcomings of XML-based query mechanisms, some modifications of XPath have been proposed. Bird et al. defined and implemented LPath [BCD+06] which extends XPath with an immediate precedence axis, subtree scoping, and edge alignment. Subtree scoping refers to the ability to restrict operations to the subtree below a node. Edge alignment is the ability to navigate to a left-most or right-most descendent of a node. Precedence in LPath is defined by the order of the terminals of an annotation hierarchy, i.e., the linear order is dependent on the hierarchical order. Lai and Bird showed that LPath$^+$, which uses the LPath extensions within Conditional XPath [Mar04] to support simple closures, is equivalent to first-order logic on trees [LB10]. LPath is translated to SQL and evaluated on an RDBMS using the following labeling scheme to store trees: Each node has two labels, l and r, which are independent for each tree. For the left-most leaf v_1 we set $v_{1.l} = 1$ and for each leaf v_i we set $v_{i.r} = v_{i.l} + 1$. For two consecutive leaves v_i and v_{i+1} we set $v_{i+1.l} = v_{i.r}$. Finally, the minimally covered value of l and the maximally covered value of r are propagated upwards for non-terminals. These two labels, along with the identifier of the parent node, allow the implementation of XPath axes and LPath extensions using simple value comparisons, similar to a pre/post-order scheme.

Vitt defined DDDquery [Vit05] which extends XPath with axes for immediate precedence and alignment of parallel text spans, as well as regular expressions over path expressions, the ability to return subtrees instead of node sets, and the ability to bind node sets within a path expression to a node variable and reference it later. The latter feature enables the user to specify cycles in the query. The DDDquery data model is not an XML tree, but a directed, acyclic graph (DAG) in which nodes may have multiple parents and the entire graph may have multiple roots. DDDquery thus supports concurrent hierarchies which are distinguished by node types. Specifically, the alignment of parallel text spans is implemented by a particular node structure which can be evaluated by dedicated axis steps. Precedence is defined according to the position of a text span in the original text, independently of any hierarchical organization. The syntax does not permit negated filter expressions and thus only supports existential queries. DDDquery is translated to SQL and evaluated on an RDBMS using a pre/post order scheme adapted to the encoding of DAGs [TL05]. Regular path expressions are implemented using the recursive query functionality of modern RDBMS, i.e., Feature T131 in [ISO03]. A heavily modified version of DDDquery was used as an intermediary step in the translation from AQL to SQL in the first implementation of Annis 2 [Ros11].

This dialect lacked many features, such as regular path expressions or the alignment of parallel texts. Instead, it included functionalities required by Annis, such as the ability to specify an edge type or edge annotations and simple boolean formulas over path expressions.

It is our experience that XPath-based query languages are ill-suited for querying multi-layer corpora. Their basic primitive, the path expression, makes it difficult to write queries that access multiple annotation hierarchies and other, non-hierarchical annotation layers simultaneously. Consider the query shown in Listing 5 against the TüBa-D/Z corpus [HKN+04]. It references two independent hierarchies, a dominance and a coreference layer. XPath only provides the means to navigate one of them. In our DDDquery implementation, we added a type attribute to the child and descendent axes, e.g., ./child[d]::* for dominance edges, in order to formulate queries using multiple hierarchies. The query also connects two sentences in both directions, first directly through the precedence operator in line 2 and then indirectly through a coreference link in line 8. The query essentially specifies a cycle in the annotation graph. XPath does not provide the means to "remember" a node and thus cannot express such cycles. Even with node variables in DDDquery, the formulation of cycles is cumbersome. Instead, query languages that allow the user to search for text spans matching certain criteria, e.g., the existence of an annotation, and to express a linguistic relationship between these spans with dedicated operators are more intuitive. In our view, such languages make it simple to construct a query iteratively and can hide many details of the underlying data model.

```
1 cat="TOP" &
2 cat="TOP" & #1 . #2               ; match two neighboring sentences
3 field="VF" & #2 _i_ #3            ; where the prefield of the second sentence
4 cat="NX" & #3 _=_ #4              ; is a noun phrase
5 pos="ART" & #4 > #5               ; dominating a definite article
6 tok=/[Dd]../ & #5 _=_ #6
7 node & #4 _=_ #7                  ; and where the noun phrase
8 node & #7 ->coreferential #8      ; has an antecedent
9 #1 _i_ #8                         ; in the first sentence
```

Listing 5: A complex Annis query referencing two hierarchical annotation layers and specifying a cycle between text spans. Adapted from [KRZZ11].

4.3 Dedicated tree query languages

One prototypical linguistic tree query language is TGrep, developed for the Penn Treebank [MMS93]. In this corpus format, syntactically parsed sentences are stored independently from each other. Consequently, relationships that cross sentence boundaries are impossible to express. TGrep provides a very rich query language based upon immediate precedence and immediate dominance as primitives. Here, precedence is defined by the order of the tokens of the syntax tree. A TGrep query consists of nodes that are referenced by their textual content or a part-of-speech annotation for tokens, or by a syntactical annotation for non-terminals. Nodes are linked by a set of required and prohibited relationships. TGrep2 [Roh05] is an almost backwards-compatible reimplementation of TGrep. It provides additional node relationship operators and other enhancements. Nodes are combined using boolean operations instead of a specifying a set of required or prohibited relationships on them. TGrep2 supports a limited form of universal quantification, however the exact semantics are not sufficiently documented. Furthermore, to create queries whose relationship structure is not a tree, globally scoped variables can be assigned to nodes and referred to later in the query. Two interesting features of TGrep2 are the ability to mark parts of the pattern as optional and to find only those sentences in the corpus which match multiple patterns at once.

Similarly, TIGERSearch [Lez02] was developed for the German treebank TIGER [BDE+04]. It also uses parse trees of sentences as its basic structure, but nodes can have multiple annotations. For example,

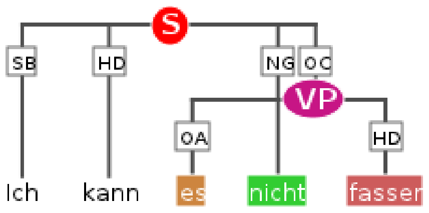


Figure 15: A syntax tree with a crossing edge caused by a discontinuous phrase.

tokens are annotated with their part-of-speech category, their lemma, and their morphological category. Edges can be labelled to denote the grammatical function of the target constituent. The corpus format supports crossing and secondary edges. The former are required because of the existence of discontinuous phrases in German, e.g., the verb phrase *es fassen* in Figure 15, which is interrupted by the word *nicht*. The latter are used to model coordinated phrases in which constituents are missing in one conjunction. For example, in Figure 16 the noun phrase *die in der Wochenendausgabe genannte Zahl* is nominally a constituent of the two verb phrases headed by the verbs *dementieren* and *bestätigen*. However, the second verb phrase is shortened and the noun phrase is only attached to first verb phrase in the syntax tree. A secondary edge, depicted as a dotted blue line, is inserted into the graph to indicate that the noun phrase is a constituent of the second verb phrase. Secondary edges are not processed by the dominance operator and must be queried separately. The corpus is converted into a Prolog-like facts database with which the query is unified during evaluation. To limit the search space, TIGERSearch first eliminates those graphs in the corpus which do not contain a match for the most restrictive positional annotation used in the query. In the original implementation variables are existentially qualified. However, there exists a reimplementation which supports universal quantification [MLV08]. Besides its widespread use in corpus linguistics, TIGERSearch has also been employed to model and search protein databases [SR03].

VIQTORYA [SK02], developed for the Tübingen Treebank [HBK$^+$00], allows the specification of multiple unconnected trees in one query. This ability is needed because the treebank contains structures which are not connected to the sentence root, e.g., disfluencies such as *uh* in English or *ähm* in German, repetitions, hesitations, and "false starts". The query language theoretically allows the construction of graphs in which there are nodes with multiple parents, but it is unclear if this is actually supported by the implementation. Explicitly numbered variables representing a token, a part-or-speech category, or a syntactical function can be combined using a direct and indirect dominance and an underspecified precedence relationship by referring to the variable number. There is no support for immediate precedence. Complex queries can be created using conjunction, disjunction, and negation. However, the latter is restricted to atomic terms, i.e., variable definitions and linguistic relationships. Variables are existentially qualified. The query is translated to SQL and evaluated on an RDBMS, using a scheme that explicitly stores tokens, non-terminals, and relationships in separate tables. Theoretically, the database schema and the query language allow annotations that cross sentence boundaries. However, tokens and relationships are indexed

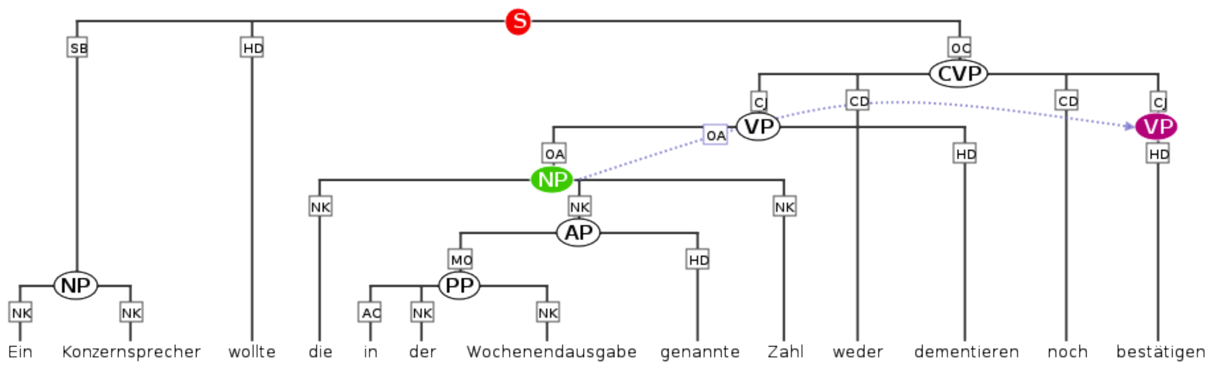


Figure 16: A syntax tree from the TIGER Treebank containing a secondary edge.

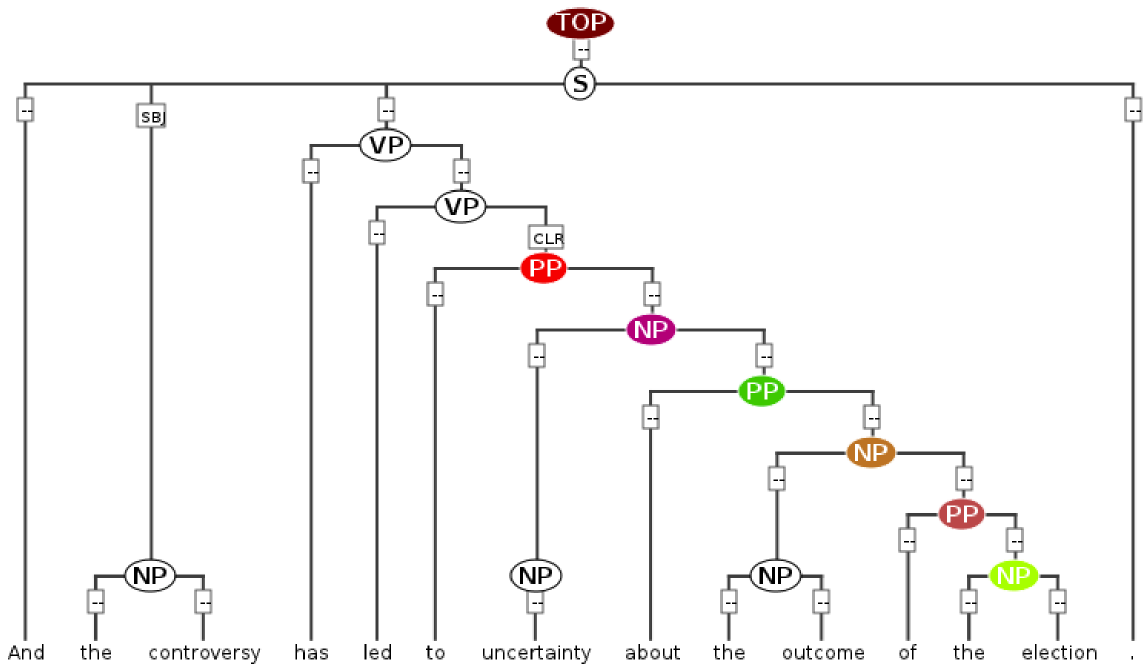


Figure 17: A syntax tree of an English sentence with nested prepositional and noun phrases.

by a tree identifier in the database and the treebank uses one tree-like structure per sentence.

Finite Structure Query (fsq) [Kep03] uses first-order logic as its language paradigm. Propositional annotations are treated as node properties and expressed using unary predicates on node variables. There exists a predicate that tests for the tokenness of a node and one that tests for the label of an edge which is pushed down to the child node. Linguistic relations such as immediate dominance, precedence, and secondary edges between nodes are expressed as binary relations. These atomic terms can be combined using a LISP-like syntax into a first-order logic formula, including implication and universal qualification. Each tree in a treebank is tested independently of the others. Therefore, annotations crossing sentence boundaries are not possible. The time to test whether a fsq formula is satisfied by a tree is exponential in the quantifier depth of the formula. Evaluation time is therefore sensitive to how a query is formulated.

Similarly, MonaSearch [MK09] is based on monadic second-order logic (MSO), i.e., first-order logic extended with set variables. MSO has the property that the transitive closure of a relation defined in MSO is also in MSO. For example, dominance can be defined as the closure of the parent-child relationship. This makes it possible to query arbitrarily nested structures. For example, the syntax tree of a sentence from the Penn Treebank [MMS93] in Figure 17 contains three levels of alternating, nested prepositional phrases (PP) and noun phrases (NP). This structure could, in principle, be extended to an arbitrary depth, only limited by the need of a sentence to be finite. A formula in first-order logic is unable to capture such a structure because it has to be extended for each additional level. In MSO, one can define a binary relation of a prepositional phrase dominating a noun phrase and then assert that the top-level prepositional phrase and the most deeply nested noun phrase are contained in the transitive closure of this relationship. MSO formulas are decidable on trees and can be efficiently evaluated by converting the formula into a tree automaton and testing each tree in the treebank if it is accepted by the automaton. This process is delegated to the external toolkit MONA [KM01], which was originally developed for hardware verification. Annotations that cross sentence boundaries are not possible. Similarly, crossing and secondary edges, such as those in Figure 15 and Figure 16, are ignored because they violate the tree structure. The translation of a MSO formula into a tree automaton is exponential in the length of the formula. However, the evaluation of the automaton on a tree is linear in the size of the tree. Maryns and Kepser argue that the latter is the dominant factor for typical linguistic queries on large treebanks and that MonaSearch is therefore linear in the size of the treebank [MK09].

4.4 Multi-layer query languages and corpus systems

Except for DDDquery, the query languages discussed so far are limited to querying tree-like structures encoding the syntax of a single sentence. Consequently, annotation structures crossing sentence boundaries cannot be queried. The corpus formats on which these languages operate also lack the ability to model overlapping annotations and concurrent hierarchies. In contrast to the large variety of linguistic tree query languages, there are few truly multi-level annotation query languages and corpus systems.

One such language is NQL [VEKC03], developed as part of the NITE XML Toolkit [CEHK05]. It is a successor of the query language Q4M, developed for the MATE Workbench annotation tool [MIM+01]. The underlying data model allows for arbitrary graphs or multi-rooted trees which are independently ordered along a structural and a temporal dimension. Each node is typed and may be associated with arbitrary key-value annotations as well as textual and timing information. The query language contains unary predicates for node properties as well as binary operators that specify how two nodes relate to each other structurally or temporally. An operator for immediate precedence is missing. Atomic terms can be combined using standard boolean operations, including implication and universal qualification. An interesting feature of the system is that queries can be chained, allowing one query to filter the results of the previous one. The query language is implemented in Java. It operates on in-memory structures that are created from XML sources and is rather slow on large data sets. There exists a batch mode to process long-running queries. Mayo et al. discuss how NQL can be evaluated more efficiently using an implementation on top of XQuery or query rewriting [MKC06].

SPLICR [RSW+09] is a web-based corpus system which also aims to provide the means to query and visualize heterogenous multi-level corpora and is, on the surface, similar to Annis. Special focuses of SPLICR are the sustainability of the stored data and a uniform access to diverse corpora. It employs ontologies of linguistic annotations which can be used to query data. The back-end implementation of SPLICR is, however, markedly different than that of Annis. It operates directly on stand-off XML files stored in an eXist database [Mei03] and uses XQuery as its query language. In a stand-off annotation format the annotation markup is detached from the original text and typically stored in a separate file [TM97]. SPLICR transforms XML files from heterogenous sources into a dedicated XML-based data model for multi-layer corpora called AnnoLab [ET07]. Queries in SPLICR can also be constructed with a graphical query builder. The corpus database is searched iteratively, i.e., the user can peruse the first results while the system is still searching. Whereas the XML-based approach proved useful with regard to long-term storage and management of heterogenous corpora, the query performance was rather slow.

4.5 Graphical query languages

An interesting alternative to textual query languages, particularly for novice users, is the visual construction of queries with the help of a graphical query builder. Such tools are available for TIGERSearch [VL02] and LPath [BL07]. The latter allows the consecutive graphical refinement of any tree found in the treebank; a process which the authors call *Query by Annotation*. The Linguist's Search Engine [RE05] has a similar approach: The user enters a sentence and the corresponding parse tree is generated automatically. They can then modify that parse tree and search for similar occurrences in data that is crawled from the web. The SPLICR corpus architecture [RSW+09] uses a graphical query builder to create XQuery expressions. In ICECUP [WN00], graphical queries are the only means to search the corpus. They can be constructed manually, created from a fulltext search with simple globbing, or derived from the structure of a previously found match. In contrast, the VIQTORYA [SK02] query tool does not offer a true graphical query builder. It can construct complex queries by listing nodes and their relations in two separate window panes.

4.6 Feature comparison

In Table 5 we compare the features of the linguistic query languages described in this section and summarize their level of support of the requirements listed in section 4.1. Every language fulfills, at least to a degree, the basic requirements of a tree query language. In particular, LPath and TGrep2 offer a very rich variety of structural operators that is not really captured by the table. A major difference between these languages is their support for boolean operations. LPath, fsq, and NQL implement full first-order logic including universal quantification. Thus, it is possible to formulate linguistic queries in these languages, which cannot be expressed in a language without universal quantification. For example, the canonical negated query used in [LB04] searches for sentences that do not include the word *saw*. MonaSearch supports a superset of first-order logic and allows users to query for arbitrarily nested structures. The languages can also be differentiated by the type of structure that can be queried by them. LPath and MonaSearch can only process trees whereas the remaining languages support DAGs.

Aside from Annis, only DDDquery and NQL meet all of the additional requirements for a query language for multi-level corpora. Indeed, the Annis data model and the use of coverage operations is influenced by NQL, whereas the dominance and precedence operators are taken from TIGERSearch. Because NQL supports first-order logic it is more expressive than Annis. However, its custom Java implementation is comparatively slow. As far as we know, Annis is faster than other linguistic query languages that support a similar feature set. A previous of Annis version used a heavily modified version of DDDquery as an intermediate language. This dialect did not implement advanced DDDquery features such as regular path expressions. In our experience, the basic language feature of DDDquery, the path expression, makes the formulation of complex queries cumbersome. Therefore, recent versions translate Annis queries directly to SQL.

Table 5: Feature comparison of linguistic query languages.

	XML-based		Dedicated tree query					Multi-level	
	LPath	DDDquery	TGrep2	TIGERSearch	VIQTORYA	fsq	MonaSearch	NQL	Annis
Linear order	Based on hierarchy	Independent	Based on hierarchy	Independent	Independent	Independent	Based on hierarchy	Based on hierarchy	Independent
Direct precedence	✓	✓	✓	✓		✓	✓		✓
Indirect precedence	✓	✓	✓	✓	✓	✓	✓	✓	✓
Limits on distance				✓					✓
Hierarchical order	XPath tree navigation	XPath tree navigation	Tree navigation	Dedicated operators	Dedicated operators	Dedicated operators	Dedicated operators	Dedicated operators	Dedicated operators
Direct (limits on children)	✓(✓)	✓(✓)	✓(✓)	✓(✓)	✓	✓	✓	✓	✓(✓)
Indirect (limits on distance)	✓	✓	✓	✓(✓)	✓	✓	✓	✓(✓)	✓(✓)
Sibling (common ancestor)	✓	✓	✓	✓					✓(✓)
Labelled edges				✓	✓	✓	✓	✓	✓
Secondary edges				✓		✓			✓
Interactions									
Edge alignment	Descendents		Descendents	Terminals					Children
Ordered children			✓						
Following/preceding sibling	✓	✓		✓					
Boolean operations	Universal FO logic	Existential FO logic	(Universal) FO logic	Existential FO logic	Value or relationship negation	Universal FO logic	Monadic SO logic	Universal FO logic	Value negation
Closures	Limited	✓					✓		
Arbitrary annotations	✓	✓		✓		✓	✓	✓	✓
Overlapping annotations		✓						✓	✓
Multiple hierarchies		✓						✓	✓
Operator scope	Single parse tree	Entire corpus	Single parse tree	Single parse tree	Single parse tree	Single parse tree	Single parse tree	Entire corpus	Entire corpus
Data model	Tree	DAG	DAG	DAG	DAG	DAG	Tree	DAG	DAG
Corpus format	Penn Treebank or similar	gXDF	Penn Treebank	TIGER corpus format	NEGRA export format	NEGRA export format	NEGRA export format	Stand-off XML	Various
Implementation	RDBMS	RDBMS	Custom	Prolog-like	RDBMS	Custom	Tree automaton	Custom	RDBMS

5 Experimental setup

In this section we describe the test data we use to measure the performance of Annis on MonetDB. We use the TIGER Treebank as a test corpus because it is one of the largest corpora containing hierarchical annotations available in Annis. It is used by students at the linguistics department of the Humboldt Universität zu Berlin which enables us to collect a real-world query workload for testing purposes. We also describe the hardware and software configuration on which we perform our measurements and the measurement procedure.

5.1 The TIGER Treebank

The TIGER Treebank version 2.1[7] [BDE+04] is a corpus containing about 900,000 tokens and 50,000 sentences of German newspaper texts annotated with syntactic structure. Tokens in the TIGER Treebank are annotated with three annotation layers. Part-of-speech information is encoded in a *pos* layer according to the Stuttgart/Tübinger Tagset [STST99], morphological information is encoded in a *morph* layer according to the TIGER morphology annotation scheme [CHSSZE05], and the token's lemma is stored in a *lemma* layer. Sentences are annotated with their syntactic structure which is encoded as a tree-like syntax graph with the tokens as leaves. Non-terminals in this graph represent phrases consisting of one or more constituents and are annotated with their constituent category in a *cat* layer. Edges in the graph are annotated with the syntactic function of the child node in a *func* layer. Both the *cat* and *func* annotation layers use the TIGER annotation scheme [A+03]. In addition to normal dominance edges pointing from a parent phrase to a child constituent, the corpus also contains secondary, possibly crossing, edges linking discontinuous constituents as they are found in German syntax. An example of such a secondary edge is show in Figure 16 on page 37. Table 6 lists the number of distinct values for each annotation layer. Apart from the annotation value `"--"` whose semantics are similar to SQL's **`NULL`** value, there is little overlap between the values of different annotation layers. Values common to multiple annotation layers are listed in Table 7.

The TIGER Treebank is loaded into Annis by converting it from the TIGER-XML format to relANNIS files using the Salt'N'Pepper converter framework [ZR10]. relANNIS files are a tab-delimited representation of the Annis database schema. Table 8 lists the row count for each source table as well as the `facts` table of the materialized schema used by the PostgreSQL implementation. Table 9 lists additional information about the TIGER Treebank representation in Annis.

Table 6: Number of distinct values for each node and edge annotation layer in the TIGER Treebank.

Annotation layer	**Distinct values**
node annotations	
cat	26
pos	54
morph	274
lemma	64,870
edge annotations	
func	44

5.2 Test queries for the TIGER Treebank

In order to devise a realistic test workload we collect queries from the Annis installation at the Humboldt Universität zu Berlin's linguistics department over a period of three months. This set includes queries

[7] http://www.ims.uni-stuttgart.de/projekte/TIGER/TIGERCorpus/

Table 7: Number of values appearing in multiple annotation layers in the TIGER Treebank.

Annotation value	**Node annotations**				**Edge annotations**
	cat	pos	morph	lemma	func
--	2		307,539	121,701	246
AA	111			1	
AP	15,116			19	
CO	385			5	
CS	5,901			2	
Gen			8	13	
NP	109,119			8	
PDS		3,193		130	
PP	91,081			2	
S	72,346			6	

Table 8: Number of tuples in each table of the TIGER Treebank in Annis.

Table	**Tuples**
corpus	1,972
corpus_annotation	1,971
text	1,971
node	1,262,014
node_annotation	3,039,170
rank	2,431,307
component	226,677
edge_annotation	2,204,630
facts	5,774,939

Table 9: General information about the TIGER Treebank in Annis.

Nodes	1,262,014
Tokens (terminals)	888,563
Non-terminals	373,451
Roots	173,257
Isolated nodes	124,541
Average number of nodes in a text	640
Average number of tokens in a text	451
Average number of nodes in a component	11
Edges	1,095,165
Duplicate edge entries in rank	1,211,601
due to dominance merging	1,158,180
due to pre/post order encoding	53,421

from students and/or researchers as well as test queries by the Annis developers. In total, there are 224 unique Annis queries against the TIGER Treebank. Some of these queries return no results or are clearly invalid. Nevertheless, we include them in the test set as they might shed light on performance issues.

5.2.1 Test query groups

A high-level overview of the queries in our test is provided in Table 10. The queries are grouped by the number of search terms and the type of linguistic operation they use. The five most common combinations are emphasized. Simple queries with only one search term and no binary operation are most common. They are followed by queries with two or three search terms connected either by coverage or by precedence operations. Queries containing dominance operations are rather rare. To get a better idea of the type of queries contained in the test set, we can divide them into six groups which we describe below.

Table 10: Test queries grouped by the number of search terms and used linguistic operations.

Search terms	**Binary linguistic operations** Coverage	Precedence	Dominance	**Number of queries**
1				**74**
2			✓	3
2		✓		**16**
2		✓	✓	1
2	✓			**71**
3			✓	4
3		✓		**15**
3		✓	✓	3
3	✓			**28**
3	✓		✓	3
3	✓	✓	✓	1
4			✓	1
4		✓	✓	2
4	✓	✓	✓	2

Group A: Simple queries with only one search term

74 queries, or almost a third of the test set, are simple queries with only one search term. These include the trivial queries **`node`** and **`tok`** as well as `lemma` which lists every node with a *lemma* annotation. Of the rest, 39 queries are annotation searches, such as `pos="VVFIN"`, and 32 queries are text searches, such as `"man"`. These queries can also be analyzed by how the text that is searched for is specified. 38 queries use an exact string match and 33 queries use a regular expression. Of these, there are 28 regular expressions in which the first character is fixed, such as `/kann.*/`, compared to five in which the first character is not fixed, such as `/[Kk]ann.*/`. Curiously, there are 17 regular expressions which match exactly one word, e.g., `lemma=/gesunken/`. Consider that Annis implicitly anchors regular expression searches at the beginning and at the end. Therefore, these queries can be substituted by an exact string match, i.e., `lemma="gesunken"`.

Group B: Two search terms linked by Exact Cover

The second largest group contains queries in which two search terms are linked by an Exact Cover operation. Most queries in this group search for specific grammatical forms of a particular verb. For example, the first query in Example 1 searches for finite forms of the verb *wachsen*. Other queries search for infinite verb forms using `pos="VVINF"`, perfect verb forms using `pos="VVPP"`, and infinites with *zu* using `pos="VVIZU"`. The second query searches for finite forms of verbs with a certain prefix. The remaining queries search for adjectives ending in *nd* followed by one or two characters, e.g., query 3, diminutives, e.g., query 4, and tokens annotated with `lemma="--"` which are not punctuation marks, e.g., query 5. The last query finds words such as *ap* or *ips*, i.e., news agency acronyms in newspaper articles. It also matches English words in German newspaper texts, or internet URLs. Similarly to group A, there are some queries which use a regular expression matching exactly one word, e.g., `pos=/VVFIN/`. In total there are 69 queries in this group.

```
1  pos="VVFIN" & lemma="wachsen" & #1 _=_ #2
2  tok=/be.+/ & pos=/VVFIN/ & #1 _=_ #2
3  pos="ADJA" & tok=/.+nd..?/ & #1 _=_ #2
4  lemma=/.+[^aeiouäöü]chen/ & pos="NN" & #1 _=_ #2
5  lemma="--" & pos!=/\\$.*/ & #1 _=_ #2
```

Example 1: Annis queries containing two search terms linked by Exact Cover.

Group C: Three search terms linked by Exact Cover

The third group contains queries in which three search terms are linked by two Exact Cover operations. The majority of these queries search for an adjective derived from a specific verb. For example, the first query in Example 2 searches for adjectives based on the verb *wachsen*. These queries utilize two token tests: `tok=/ge.+en..?/` in which the first character is fixed and `tok=/.+nd..?/` in which the first character is unknown. The remaining queries in this group search for perfect forms of a specific verb, e.g., query 2, and nouns with an umlaut in the plural form, e.g., query 3. In total there are 28 queries in this group.

```
1  pos="ADJA" & tok=/ge.+en..?/ & #1 _=_ #2 & lemma=/(ge)?wachsen/ & #1 _=_ #3
2  pos="VVPP" & tok=/ge.+en/ & #1 _=_ #2 & lemma=/(ge)?kommen/ & #1 _=_ #3
3  lemma=/[^äöü]+/ & tok=/.+[äöü].+/ & pos="NN" & #1 _=_ #2 & #2 _=_ #3
```

Example 2: Annis queries containing three search terms linked by Exact Cover.

Group D: Two search terms linked by precedence

The queries in the fourth group consist of two search terms which are linked by a precedence operation. These can be fairly specific, such as the first query in Example 3 searching for the phrase *müssen weg*, or rather unspecific, like the second query searching for finite verbs followed closely by a noun. Note that this group contains queries that produce a large number of results, such as the last two queries. We believe that these queries are based on faulty assumptions about the Annis data model and discuss them further in section 5.2.2. In total there are 16 queries in this group.

```
1  lemma="müssen" & lemma="weg" & #1 . #2
2  pos="VVFIN" & pos="NN" & #1 .1,3 #2
3  pos=/VM.*/ & pos=/VV.*/ & #1 .* #2
4  lemma="müssen" & pos!=/VV.*/ & #1 .* #2
```

Example 3: Annis queries containing two search terms linked by precedence.

Group E: Three search terms linked by precedence

The fifth group contains queries in which three search terms are linked by two precedence operations. Most of these queries either return no results, e.g., query 1 in Example 4, or a very large result set, e.g., query 2 which returns more than 60 million matches. Similarly to group D, we believe that these queries are based on faulty assumptions about the Annis data model. Indeed, the only meaningful query in this group is the last one below, searching for subordinate clauses with the subject *man* and the reflexive pronoun *sich*. In total there are 15 queries in this group.

```
1  pos=/VM.*/ & pos=/VV.*/ & pos="($.|$,)" & #1 .* #2 & #2 .* #3
2  lemma="müssen" & pos!="VV.*" & pos="$." & #1 .* #2 & #2 .* #3
3  pos="KOUS" & "man" & "sich" & #1 . #2 & #2 . #3
```

Example 4: Annis queries containing three search terms linked by precedence.

Group F: Queries with a variable number of search terms and mixed operations

The remaining 22 queries consist of up to four search terms which are connected by various binary operations. 20 of these contain at least one dominance operation, such as the first query in Example 5, searching for sentences and their finite verb. The remaining two queries consist of two search terms connected by an Inclusion operation. This group contains the most complex queries in the test set. For example, the query in Example 6 consists of four search terms, one unary linguistic constraint, and six binary operations. It searches for sentences with two constituents in the prefield, i.e., appearing before the finite verb at the beginning of the sentence.

```
1  cat="S" & pos="VVFIN" & #1 >* #2
2  cat="S" & pos="VVFIN" & #1 _i_ #2
```

Example 5: Annis queries with a variable number of search terms and mixed operations.

```
cat="S" & #1:root &
pos="VVFIN" & #1 >[func="HD"] #2 &
cat & #1 > #3 & #1 _l_ #3 &
cat & #1 > #4 & #3 . #4 & #4 . #2
```

Example 6: Complex Annis query containing coverage, precedence, and dominance operations.

5.2.2 Invalid test queries

About 15% of the queries contain some kind of error. We keep these queries in the test set because they can help to illuminate performance issues in our implementation. Most of these are trivial mistakes; in many cases a keyword or annotation value is misspelled or the wrong kind of quotes is used for regular expression searches:

Unknown annotation layer. Some annotation searches use an annotation name that does not exist in the TIGER Treebank, e.g., `token=/.+nd..?/`. In this case the user most likely meant a text search.

Misspelled annotation value. Some annotation searches use an annotation value which does not exist in the TIGER annotation scheme, but is similar to a known value, e.g., `pos="ADJ"`. Most likely the user meant the `"ADJA"` annotation for adjectives.

Regular expressions in an exact string search. Some exact string searches contain regular expression special characters, e.g., `pos!="VV.*"`. Most likely the wrong type of quotes are used in this case.

Badly escaped regular expression searches. Some regular expression searches do not escape special character correctly, e.g., `pos!=/$.*/`. To search for punctuation marks the user has to escape the `$` character with two backslashes, e.g., `/\\$.*/`. We correct these queries in our test set and always escape the `$` character.

Other errors indicate a misunderstanding of the Annis data model by the user:

Indirect precedence ignores sentence boundaries. The query `pos=/VM.*/ & pos=/VV.*/ & #1 .* #2` searches for modal verbs followed by a full verb. However, this search returns many pairs of unrelated verbs since the indirect precedence operator does not stop at sentence boundaries. Most likely the user assumed that the precedence operator is only defined on the tokens of a single sentence; this is the behavior of the precedence operator in TIGERSearch. In order to find pairs of related modal and full verbs the user has to explicitly restrict the search to a sentence, e.g., use the query `cat="S" & pos=/VM.*/ & pos=/VV.*/ & #1 > #2 & #1 > #3 & #2 .* #3`.

Using punctuation marks to limit precedence to sentences. Similarly to the example above, the query `pos=/VM.*/ & pos=/VV.*/ & pos="$." & #1 .* #2 & #3 .* #3` searches for a modal verb followed by a full verb which in turn is followed by a sentence punctuation mark. However, this search also returns many unrelated results, almost 20 million in the TIGER Treebank. Most likely the user realized that the precedence operator in Annis does not stop at sentence boundaries and wanted to limit it by explicitly specifying the end of the sentence in the query.

No universal qualification of negation. Our interpretation of the query `cat="S" & pos=/VM.*/ & pos!=/VV.*/ & #1 >* #2 & #1 >* #3` is that the user intended to search for sentences containing a modal verb, but no full verb. However, such a query cannot be expressed in Annis because negated searches are existentially qualified. The query above will match every sentence containing a modal verb and for each token in such a sentence return a pair consisting of the modal verb and that token unless is is a full verb.

5.3 Test systems

The main test system on which we measure the performance of Annis on MonetDB is a sysGen Super-Server[8] with two six-core Intel Xeon Westmere 2.66 GHz processors[9], 48 GB RAM, and three 15000 rpm hard disks in a RAID 1 configuration. This server was specifically purchased to run an Annis installation used by the researchers of the SFB 632. The second test system is a mid-2010 Apple Macbook Pro[10] with an Intel Core 2 Duo 2.4 GHz processor[11], 4 GB RAM, and a 5400 rpm hard disk. Both computers run a version of Ubuntu Linux. Additional technical information about them can be found in Table 11.

We use these two machines to represent two different usage scenarios of Annis. In the first use case, multiple users access a centrally located server, e.g., in an educational context. Performance is crucial in this setting because many users access the system concurrently and they may be especially dissatisfied with an overly long waiting period. In the other scenario a researcher is working on his own data on his own machine. We also strive for good performance in this setting, but the demands are not as strong. A researcher may be better able to compensate for wait time and may not have access to high-end hardware in any case.

[8] http://www.sysgen.de/
[9] http://ark.intel.com/products/47922/Intel-Xeon-Processor-X5650-(12M-Cache-2_66-GHz-6_40-GTs-Intel-QPI)
[10] http://support.apple.com/kb/SP583?viewlocale=en_US
[11] http://ark.intel.com/products/35568/Intel-Core2-Duo-Processor-P8600-(3M-Cache-2_40-GHz-1066-MHz-FSB)

Table 11: Hardware specification of the two test systems.

	Server	**Laptop**
Manufacturer and name	sysGen SuperServer	Apple Macbook Pro rev. 7,1
Processor(s)	2x Intel Xeon Westmere X5650 2.66 GHz	Intel Core 2 Duo 2.4 GHz
CPU cores per processor	6 cores with hyper-threading	2 cores
CPU cache per processor	12 MB	3 MB
Main memory	48 GB DDR3 1333 MHz	4 GB DDR3 1066 MHz
Hard disk(s)	3x 300 GB SAS2 15000 rpm	250 GB SATA 5400 rpm
Hard disk cache	16 MB on disk / 512 MB on controller	8 MB
Hard disk configuration	RAID 1	
Purchase date	November 2011	January 2011
Operating system	Ubuntu Linux 11.10	Ubuntu Linux 12.04 LTS

5.4 MonetDB configuration

We use the Jul2012 release of MonetDB, i.e., version 11.11.5, and compile it from sources using the default build options. MonetDB always uses the entire available memory. Unless otherwise noted, we do not change any database runtime properties. The databases are accessed in read-write mode, using MonetDB's default optimizer pipeline, and 12 threads.

5.5 PostgreSQL configuration

We use PostgreSQL version 9.1.4 and compile it from sources using the default build options. There are two configuration options which mainly influence the memory usage by PostgreSQL:

shared_buffers Specifies the size of the buffer cache. On a dedicated database server, the PostgreSQL documentation recommends to use 25% of the available memory [Pos12]. On the server, we set shared_buffers to 10 GB. This amount is big enough to hold the entire database in main memory. If a lower setting is used, e.g., 1 GB, queries will interfere with each other, i.e., they will be slower if multiple queries are tested in a random order compared to when they are run consecutively. This effect vanishes if 10 GB is used for the buffer cache. On the laptop, we used 1 GB.

work_mem Specifies the amount of memory PostgreSQL uses for a single sort or hashing operation before temporarily writing data to disk. A query plan may contain many such operations. Therefore, the amount of memory needed for one operation depends on the complexity of the queries and the number of concurrent users. One the server, we set work_mem to 1 GB. At this size, we did not notice any operation using the hard disk. On the laptop, we used 128 MB.

A third memory-related option, effective_cache_size, is used to estimate the size of the operating system disk cache. The query planner takes this setting into account while calculating the cost of an index scan. The PostgreSQL documentation recommends to set this value to 75% of the available memory. On the server, we set it to a slightly lower value of 30 GB because there are other PostgreSQL instances running at the same time. However, they do not execute any queries. On the laptop, we set it to 3 GB. We use the default values for other PostgreSQL configuration options.

5.6 Measurement procedure

We report runtimes as measured by the Annis client. To evaluate the performance of individual queries, we run each query 11 times, discard the first run, and compute the mean of the remaining ten runs. We do not count the first run in order to ensure that the data required to evaluate a query has already been loaded into main memory during the measurement and is not displaced by a previous query. To measure the overall performance of a system, we create various workloads consisting of 10000 queries each. A workload is generated by randomly choosing a query from the test set, taking into account their original distribution. We do not discard the runtime of any query in a workload. Instead, in order to ensure that the database is loaded into main memory, we run every query from the test set once before running the workload.

6 Port of Annis to MonetDB

In this section we first describe the necessary modifications of the SQL code generated by Annis in order to evaluate Annis queries on MonetDB. We then identify various performance bottlenecks and develop strategies to remedy them. Specifically, we optimize the query execution plans generated by MonetDB, improve the performance of string searches in general and regular expression searches in particular, and remove unnecessary operations from the SQL query. At the end of the section, we measure the impact of each optimization and compare the performance of Annis on MonetDB to the implementation on PostgreSQL.

6.1 Initial port of Annis to MonetDB

Only a few changes to the generated SQL code are required in order to evaluate Annis queries on MonetDB. Clearly, the implementation of regular expression searches has to be adapted because it uses a PostgreSQL-specific operator. Aside from this, two additional minor modifications are necessary: The evaluation of boolean attributes and the treatment of reserved words in SQL.

6.1.1 Regular expression pattern matching

MonetDB does not directly support the evaluation of regular expressions in SQL statements. Neither the **SIMILAR TO** nor the **LIKE_REGEX** predicates are implemented. However, the underlying MAL kernel contains a function called `pcre.match` which matches the string in its first argument against a regular expression in its second argument using the freely available PCRE library [H+97]. This MAL function can be exposed by defining a SQL function wrapper as follows:

```
CREATE FUNCTION pcre_match(s STRING, pattern STRING)
  RETURNS BOOLEAN
  EXTERNAL NAME pcre.match;
```

The text search `/[Dd]as/` can then be implemented using the following SQL fragment:

```
pcre_match(node.span, '^[Dd]as$')
```

The `pcre.match` function returns **FALSE** if the tested value is **NULL**. Thus, in order to correctly evaluate a negated regular expression search, e.g., `tok!=/[Dd]as/`, an additional **IS NOT NULL** predicate is necessary.

```
node.span IS NOT NULL AND
NOT pcre_match(node.span, '^[Dd]as$')
```

6.1.2 Evaluation of boolean attributes

The PostgreSQL implementation uses boolean value expressions of the form `attribute IS TRUE` and `attribute IS FALSE` to evaluate boolean attributes. These predicates are not supported by MonetDB. Instead, a comparison with a literal truth value of the form `attribute = true` and `attribute = false` has to be used. We have changed the PostgreSQL implementation to also use a comparison with a boolean literal in order to improve the consistency of the generated SQL code between both implementations.

6.1.3 Reserved keywords as attribute names

The strings `left` and `right` are reserved keywords in SQL and must be quoted if they are used as attribute names. PostgreSQL is lenient insofar as it allows to use them unquoted if they are qualified with a table name, e.g., `node.left`. In MonetDB, this produces a syntax error. To retain these reserved keywords as attribute names they have to be quoted, e.g., `node."left"`.

6.2 Query execution plans

Even simple Annis queries may perform slowly on MonetDB, if the corresponding SQL query is generated according to the process described in section 3.3.6. Furthermore, the runtime may be highly dependent on how the query is formulated, e.g., the order of the search terms. This is demonstrated by the first query in Table 12 which searches for sentences and the finite verb they contain. In the second query, the order of the search terms has been switched. The second column, labeled **FROM**, lists the runtime in seconds of the *COUNT* query function on the initial port of Annis on MonetDB. We will explain the remaining columns below. Clearly, a performance of 13 seconds or even 27 seconds is not acceptable.

Table 12: Runtime (in seconds) of a simple Annis query using different query execution plans.

Annis query	Nested		CTE
	FROM	**WHERE**	**strategy**
`cat="S" & pos="VVFIN" & #1 _i_ #2`	12.6	27.4	1.0
`pos="VVFIN" & cat="S" & #2 _i_ #1`	27.3	12.8	1.0

6.2.1 Computation of query solutions in a nested subquery

Listing 6 contains a template for a SQL query implementing the *COUNT* query function of an Annis query q with n search terms. It combines the template to compute the solutions to an Annis query, which is shown in Listing 2 on page 26, with the template for query functions, which is shown in Listing 4 on page 29. We call this template the *nested/FROM template*, because the solutions to the Annis query are computed as a nested subquery in the SQL code implementing the *COUNT* query function, and because the tables that are required to retrieve the information of an individual text span are joined in the **FROM** clause. The execution plan that MonetDB generates according to the nested/FROM template for the first query in Table 12 is shown in Figure 18. The plan only contains selections and joins. Other operations such as projections and aggregations have been omitted for simplicity. Each leaf shows the number of tuples in the accessed table. Each inner node shows the number of tuples produced by the operation as well as the largest intermediate result next to the predicate that produced it. First, key-value annotations matching the *second* search term are selected from the `node_annotation` table and are joined with the `node` table to produce candidates for the second search term. Then, the `node` table is joined again using the predicates implementing the Inclusion operator. Finally, tuples matching the *first* search term are selected from the `node_annotation` table and joined to produce the solutions to the query. This join order is not optimal because it unnecessarily produces a very large intermediate result during the computation of the second join. A better alternative would be to compute candidates for each search term individually before computing the join that implements the Inclusion operator. This strategy would reduce the size of the input on the right-hand side of the Inclusion join from 1,262,014 tuples to 72,346 tuples.

There are various ways to influence how MonetDB executes SQL queries generated according to the nested/FROM template. If we switch the order of the search terms, i.e., evaluate the second query in Table 12, the change will be reflected in the execution plan. This will double the size of the intermediate result in the Inclusion join from roughly 64 million to 129 million and also double the runtime of the query. Curiously, if we move the predicates joining the accessed tables by their foreign key reference from the **FROM** clause to the **WHERE** clause, the order in which the search terms are evaluated is reversed in the execution plan. This change is reflected in the runtime reported in the third column of Table 12. We call this SQL generation template the *nested/WHERE template*. We also observe, for other queries, that the order in which the predicates are listed in the **WHERE** clause influences the execution plan chosen by MonetDB. For example, predicates on the attributes `component.name` and `component.type` are evaluated in the order they appear in the SQL query if the attribute `component.name` is not tested for a **NULL** value. However, the attribute `node_annotation.name` is always evaluated before the attribute `node_annotation.value`, regardless of their order in the SQL query. Due to the large number of possibilities we cannot test every

```
SELECT count(*)
FROM   (
         SELECT DISTINCT node1.id AS id1, ..., nodeN.id AS idN
         FROM   node AS node1 JOIN additional tables required to evaluate the first search term,
                ...,
                node AS nodeN JOIN additional tables required to evaluate the n-th search term
         WHERE  predicates on attributes of the selected tables to evaluate the query
       ) AS solutions
```

Listing 6: SQL template of the *COUNT* query function using a nested subquery.

permutation of predicates exhaustively. We do not observe that the order in which the tables are listed in the **FROM** clause influences the execution plan, but also cannot test every possible combination. In any case, the influence on the execution plan by the variations described above is very unpredictable. We are not able to produce a desired execution plan, in which the computation of candidates for each search term is pushed down and the join implementing the Inclusion operator is evaluated last.

From our experience with PostgreSQL we would not expect any of the changes described above to have an impact on the query plan because they do not change the estimated costs of a particular operation. However, MonetDB does not use cost-based query optimization. Thus, it is more dependent on the specific formulation of the SQL query.

6.2.2 Computation of query solutions in a common table expression

We can force MonetDB to push down the operations that produce candidates for a search term by a more radical rewrite of the SQL query using common table expressions, or CTEs, in a **WITH** clause. Listing 7 contains the template of a SQL query computing the *COUNT* query function using common table expressions. We call this template the *CTE template*. For each search term in the Annis query, we create a CTE to produce candidates for the search term, e.g., in lines 2 through 6 for the first search term. The **WHERE** clause of such a CTE contains predicates on the `node.span` attribute for text searches or on the `node_annotation.name` and `node_annotation.value` attributes for annotation searches. The CTE also contains predicates implementing a unary linguistic constraint referencing the search term. Finally, if the search term is used in a dominance or pointing relationship operation, we also include predicates on the `component.type` and `component.name` attributes. In the **SELECT** clause, we list the `node.id` attribute and any other attribute that is required to implement a binary linguistic constraint referencing the search term, e.g., `node.text_ref` for coverage and precedence operations. The solutions to the query are computed in an additional CTE in lines 13 to 17, which lists the search term CTEs in its **FROM** clause and lists the predicates implementing the binary linguistic constraints of the Annis query in its **WHERE** clause. Finally, the query function is implemented in the SQL query by referencing the solution CTE outside the **WITH** clause, e.g., lines 18 an 19.

The CTE template exploits the fact that common table expressions are optimized by MonetDB in isolation. Because the solution CTE references the search term CTEs in its **FROM** clause, their results have to be computed by the time the solution CTE is evaluated. This ensures that the generation of search term candidates is pushed to the bottom of the execution plan, as shown in Figure 19 for the first query in Table 12. The size of the largest intermediate result is reduced by an order of magnitude. This difference is reflected in the runtime of the query which is listed in the last column of Table 12. For this query, the CTE template always evaluates the search term `cat="S"` first, regardless of the order of the search terms in the query.

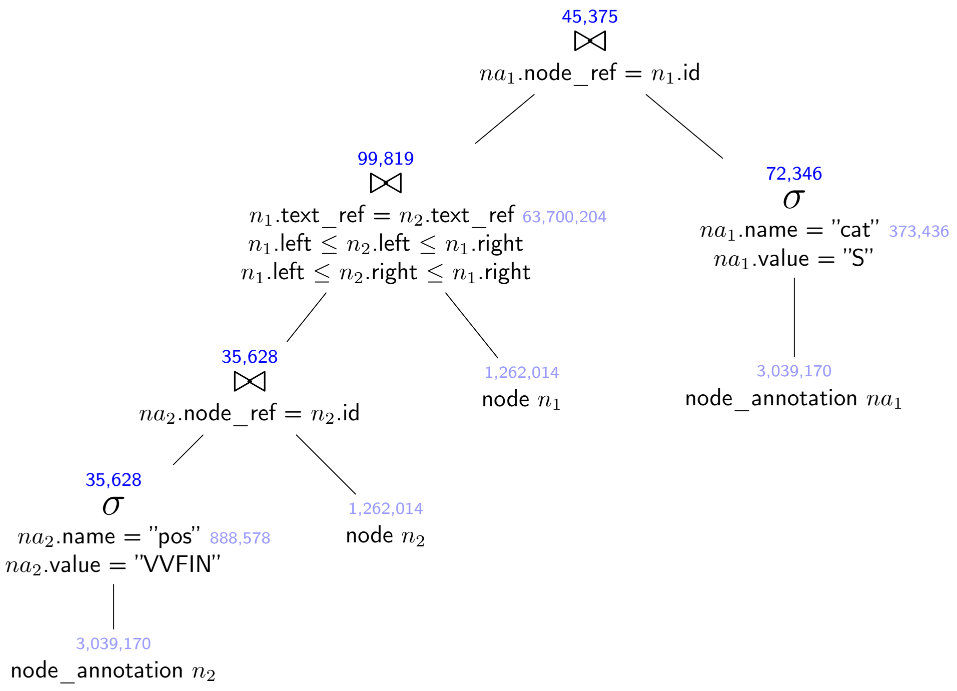

45,375
$\bowtie$
na_1.node_ref = n_1.id
99,819
$\bowtie$
n_1.text_ref = n_2.text_ref 63,700,204
n_1.left $\leq$ n_2.left $\leq$ n_1.right
n_1.left $\leq$ n_2.right $\leq$ n_1.right
72,346
σ
na_1.name = "cat" 373,436
na_1.value = "S"
35,628
$\bowtie$
na_2.node_ref = n_2.id
1,262,014
node n_1
3,039,170
node_annotation na_1
35,628
σ
na_2.name = "pos" 888,578
na_2.value = "VVFIN"
1,262,014
node n_2
3,039,170
node_annotation n_2

Figure 18: Query execution plan of an Annis query using the nested/FROM template.

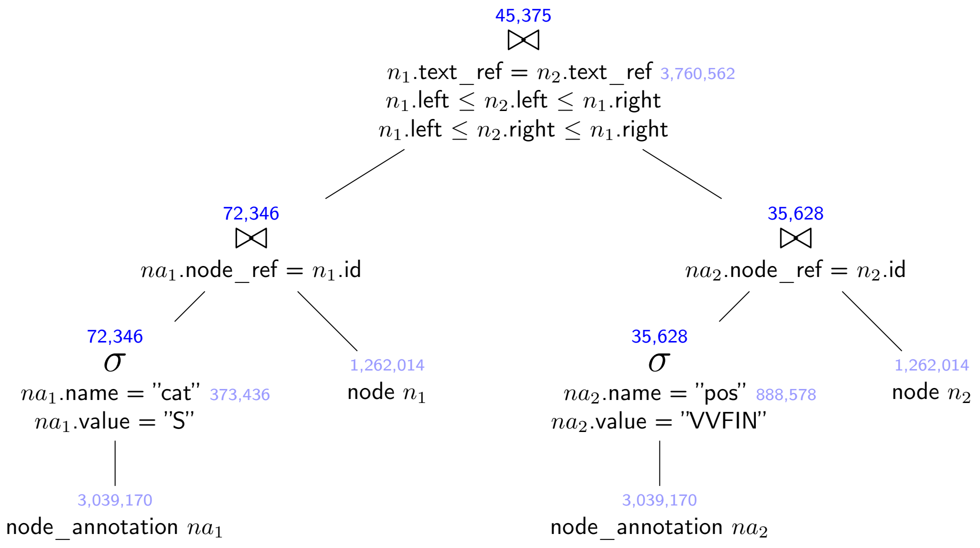

45,375
$\bowtie$
n_1.text_ref = n_2.text_ref 3,760,562
n_1.left $\leq$ n_2.left $\leq$ n_1.right
n_1.left $\leq$ n_2.right $\leq$ n_1.right
72,346
$\bowtie$
na_1.node_ref = n_1.id
35,628
$\bowtie$
na_2.node_ref = n_2.id
72,346
σ
na_1.name = "cat" 373,436
na_1.value = "S"
1,262,014
node n_1
35,628
σ
na_2.name = "pos" 888,578
na_2.value = "VVFIN"
1,262,014
node n_2
3,039,170
node_annotation na_1
3,039,170
node_annotation na_2

Figure 19: Query execution plan of an Annis query using the CTE template.

```
WITH
  span1 AS (
    SELECT node.id, other attributes required to evaluate binary relations
    FROM   node JOIN additional tables required to evaluate the first search term
    WHERE  predicates to produce candidates for the first search term
  ),
  ...,
  spanN AS (
    SELECT node.id, ...
    FROM   node JOIN ...
    WHERE  predicates to produce candidates for the n-th search term
  ),
  solutions AS (
    SELECT DISTINCT span1.id AS id1, ..., spanN.id AS idN
    FROM   span1, ..., spanN
    WHERE  predicates to evaluate binary relations
  )
SELECT count(*)
FROM   solutions;
```

Listing 7: SQL template of the $COUNT$ query function using common table expressions.

6.2.3 Performance of different query plans

Below we compare the runtime of individual queries using the nested/FROM template, illustrated by red circles, the nested/WHERE template, illustrated by red diamonds, and the CTE template, illustrated by black diamonds. The charts are ordered according to the query index in appendix B.1. In the case of the nested templates, moving the table joins from the **FROM** clause to the **WHERE** clause in the SQL query often affects the generated join plan, although this change might not be reflected in the runtime of the query. For the CTE template it does not matter whether the tables of a single span CTE are joined in the **FROM** or the **WHERE** clause. In both cases the same join plan is produced.

Group A

The graph of the first group in Figure 20 shows little or no difference in the performance of SQL queries generated using the nested templates or the CTE template. Indeed, all of the templates generate the same execution plan, except for two projections that are inserted in the execution plan generated by the CTE template to model the **SELECT** clauses of the search term CTE and the solution CTE. At the MAL layer these additional operations disappear. Here the templates only differ in how BAT variables are named. Thus, any difference in performance in Figure 20 is due to noise. This result is not surprising given the simplicity of these queries. A text search requires only one selection predicate on a single table and therefore no variation is possible in the execution plan. An annotation search joins two tables via a foreign key reference. In MonetDB this join is implemented using a precomputed hash index. This index fixes the join order of the two tables in the execution plan.

Still, there is considerable variation in the runtime of the queries in the first group. The major distinction is between queries on the left side using an exact string match, which require less than 100 ms to evaluate, and queries on the right side using a regular expression match, which require between two and three seconds to evaluate. The three queries in the middle fall outside this classification. These are the trivial queries **node** and **tok** which select every span or every token in the annotation graph respectively and the query `lemma` selecting every span with a *lemma* annotation.

The queries labeled A contain a text search with a high selectivity, i.e., less than about 6,000 matches. They require between 4 and 7 milliseconds to evaluate. A second group of text searches, labeled A^*, is

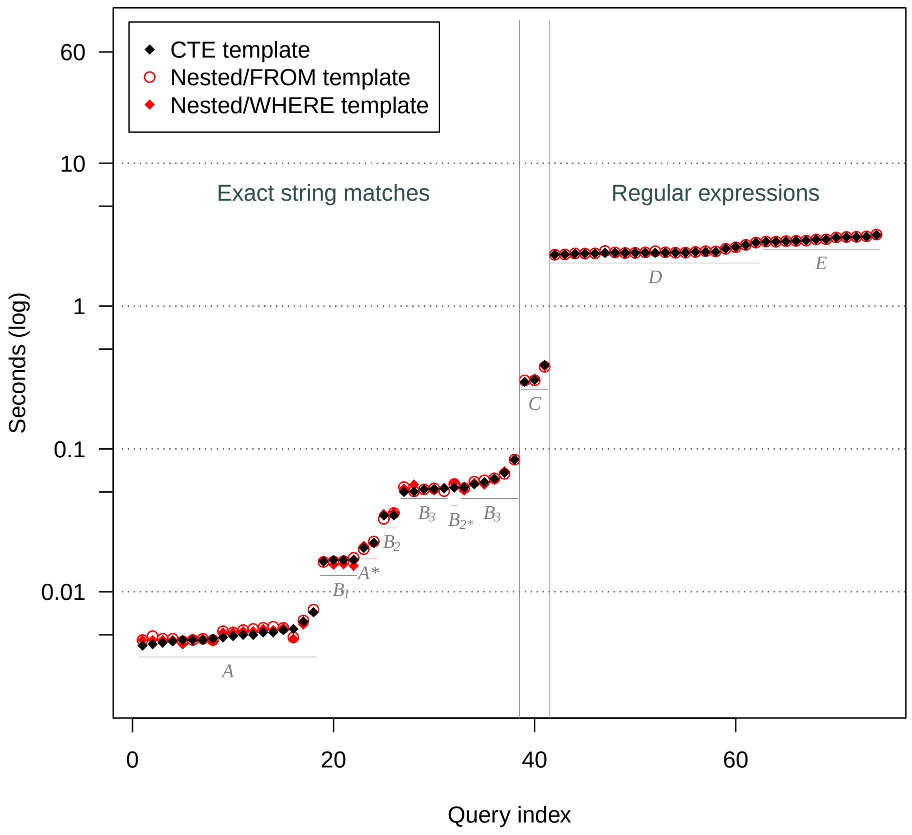


Figure 20: Performance of queries in group A depending on the SQL template.

considerably slower. They search for the definite articles *die* and *der* which are the two most common words in the corpus with about 25,000 occurrences each. These queries require about 20 milliseconds to evaluate. The performance of annotation searches mainly depends on the selectivity of the annotation key. The annotation searches labeled B_1 referencing the non-existant *word* annotation layer require only 17 milliseconds, which is faster than some text searches even though an additional table has to be joined. The selectivity of the *cat* annotation key is 0.12. *cat* annotation searches require between 34 milliseconds if they produce no results, labeled B_2, and 54 milliseconds for the query labeled B_{2*} producing about 70,000 results. Finally, *pos* and *lemma* annotation searches, labeled B_3, require between 50 ms and 84 ms. Both annotation keys have a selectivity of 0.29. The evaluation time of these annotation searches generally increases with the number of returned matches. On the right side of the graph the situation is reversed. Annotation searches using a regular expression, labeled D, require between 2.3 and 2.8 seconds to evaluate. Regular expression text searches, labeled E, are somewhat slower and require between 2.8 and 3.1 seconds to evaluate.

Group B

The graph of the second group of queries in Figure 21 also shows a similar performance of the nested and the CTE templates. However, in many cases the executed query plans are different. Typically, the nested templates create a plan that is similar to the one shown in Figure 22 for the query labeled A. First, the `node` and `node_annotation` tables are joined to compute candidates for one search term of the query. Then, the `node` table is joined again to evaluate the Exact Cover operator. Lastly, the `node_annotation` table is joined again to evaluate the second search term. MonetDB is able to evaluate the three equality predicates implementing the Exact Cover operator effectively in one MAL operation. In the TIGER Treebank, about 1.3% of the spans cover the same text as another span. The ratio is even smaller in the example: Only 13 spans, less than 0.1%, have the same extent as a span annotated with `pos="VVPP"`.

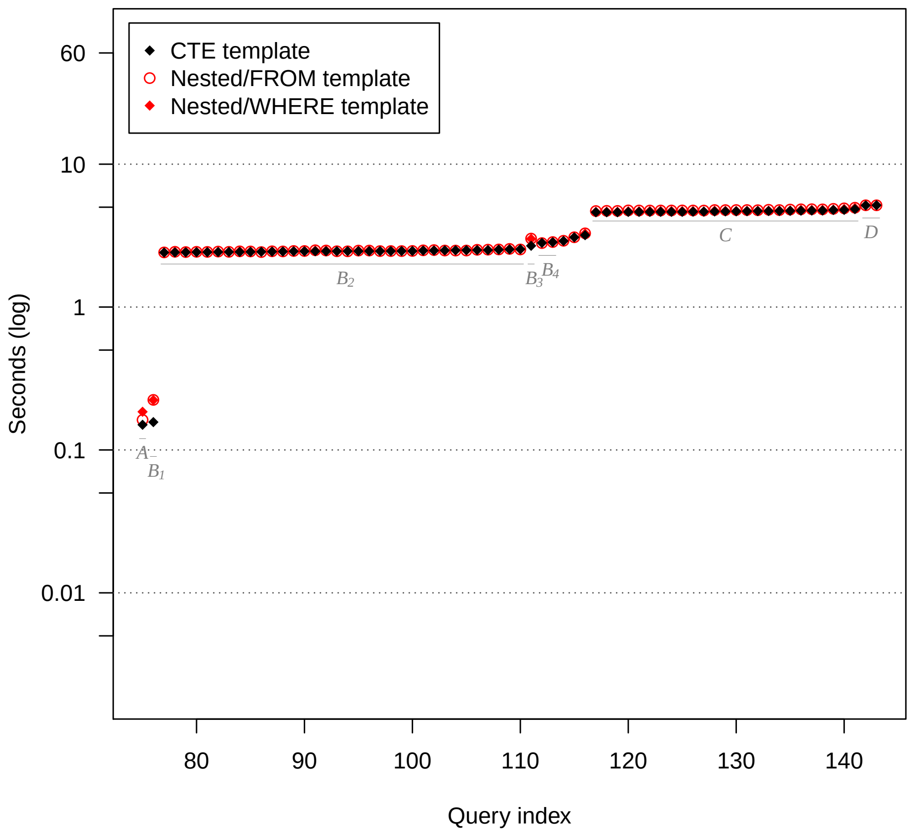


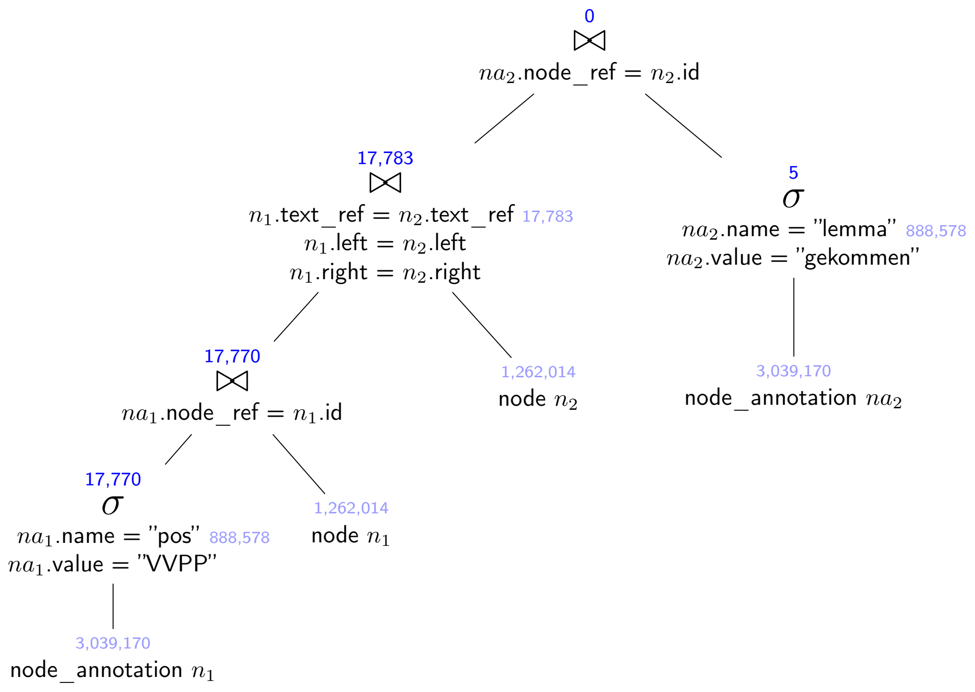
Figure 21: Performance of queries in group B depending on the SQL template.

Therefore, the largest intermediate result of this join contains only marginally more tuples than the number of candidates for the search term on the left-hand side of the join. This moderately slow growth of the intermediate result size is why the performance of the nested templates is similar to the one of the CTE template. In contrast, to implement the Inclusion operator in Figure 18, MonetDB joined the `text_ref` attribute with itself without further restricting the result set, which increases the size of intermediate results by a factor of more than 1,700.

The nested/FROM and the nested/WHERE templates generate the same query plan for the four Annis queries containing a text search. This search term is always evaluated last. Therefore, the second join of the `node_annotation` table is omitted. In effect, similarly to the CTE template, this query plan computes candidates for every search term before they are joined by a linguistic constraint. Indeed, using a Welsh t-test with $n = 10$ and $p < 0.05$, the difference in the runtime of the nested templates and the CTE template is not significant.

The nested templates also generate the same query plan for queries containing one exact string match and one regular expression search, labeled B_{1-4} in Figure 21. The exact string match, typically a *pos* annotation search, is evaluated first and the regular expression search, typically a *lemma* annotation search, is evaluated second. Except for the query labeled B_1, which contains a regular expression search on the non-existant *token* annotation layer, and the query labeled B_3, which contains a negated regular expression search on the *pos* annotation layer, there is no significant difference in the runtime of the nested and the CTE templates. The runtime of the queries labeled B_2 is fairly constant. These queries contain a regular expression matching exactly one word, e.g., `lemma=/kommen/`. The regular expressions used in the queries labeled B_4 do not have a fixed prefix. These queries are somewhat slower.

If the query contains either two exact string matches, e.g., the query A, or two regular expressions, e.g., the queries labeled C, the nested/WHERE template evaluates the search terms in the order they appear in the query. Conversely, the nested/FROM template evaluates the second search term first, as we have already seen in section 6.2.1. In the queries labeled C, the first search term is a highly selective *lemma* annotation search and the second search term is a *pos* annotation search with a comparatively lower selectivity. Consequently, the nested/WHERE template is on average 50 milliseconds faster than the nested/FROM template. The CTE strategy is even faster than that by about 40 milliseconds on average. The situation is reversed for the query labeled A. The queries labeled D contain a text search and as we have discussed above, there is no significant difference in the runtime of these queries.



0
⋈
na_2.node_ref = n_2.id
17,783
⋈
n_1.text_ref = n_2.text_ref 17,783
n_1.left = n_2.left
n_1.right = n_2.right
5
σ
na_2.name = "lemma" 888,578
na_2.value = "gekommen"
17,770
⋈
na_1.node_ref = n_1.id
1,262,014
node n_2
3,039,170
node_annotation na_2
17,770
σ
na_1.name = "pos" 888,578
na_1.value = "VVPP"
1,262,014
node n_1
3,039,170
node_annotation n_1

Figure 22: Query execution plan for query 75 using the nested/WHERE template.

Group C

The query execution plans generated by the nested and the CTE templates for the queries in group C are very similar. Consequently, there is little difference in their performance, which is shown in Figure 23. The query labeled A contains a *pos* annotation search using an exact string match, a regular expression text search, and a *lemma* annotation search using another exact string match, in this order. The CTE and the nested/WHERE template evaluate the search terms in the order they appear in the query, but the nested/WHERE template evaluates the second Exact Cover operator before evaluating the third search term. However, the runtime of both strategies is the same. The nested/FROM template evaluates the highly selective *lemma* annotation search first. Consequently, is is faster than the nested/WHERE or the CTE templates by about 80 milliseconds.

The queries labeled B contain a *pos* annotation search using an exact string match, a regular expression text search, and a regular expression *lemma* annotation search. The nested/FROM and nested/WHERE templates generate the same query plan. The search terms are evaluated in the order they appear in the query, and second Exact Cover operator is evaluated before the third search term. The CTE template also evaluates the search terms in order, but the second Exact Cover operator is evaluated last. The CTE template is on average about 90 milliseconds faster than the nested strategies.

Finally, the query labeled C contains a regular expression *lemma* annotation search, a regular expression text search, and a *pos* annotation search using an exact string match. These search terms are similar to those found in the queries labeled B, but their order is different. Again, the CTE template evaluates the

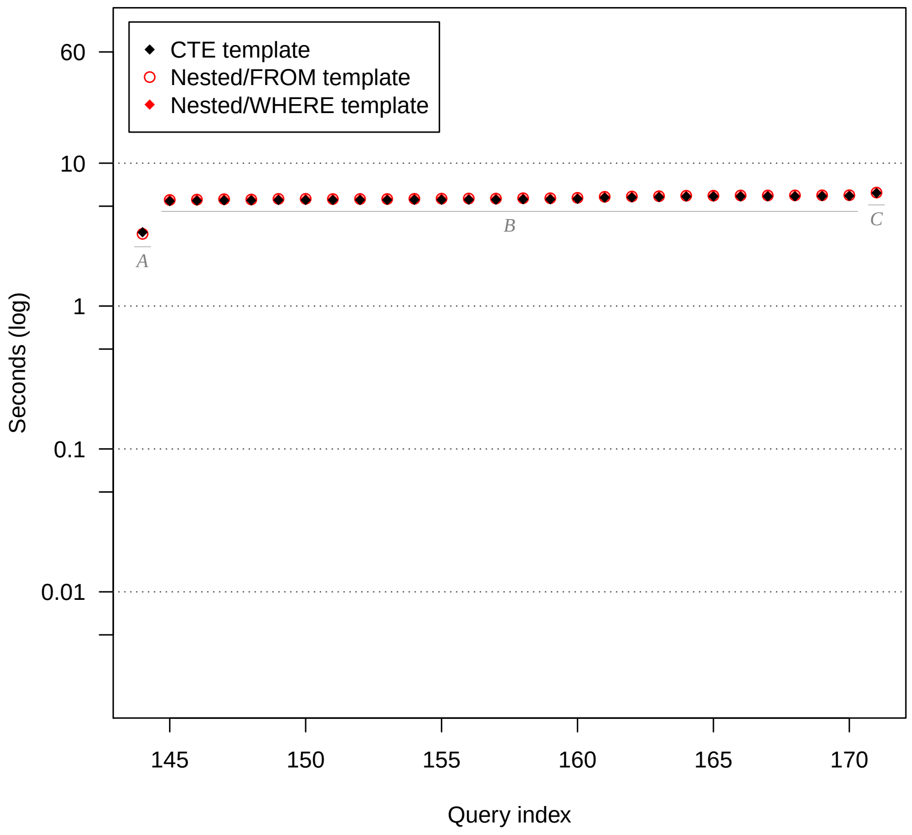


Figure 23: Performance of queries in group C depending on the SQL template.

search terms in the order they appear in the query. However, the nested/FROM and nested/WHERE templates both use the same order as for the queries labeled B, i.e., the *pos* annotation search is evaluated first. The CTE template is about 45 milliseconds faster than the nested templates.

Group D

So far, the choice of the SQL generation template has had little or no effect on query performance. However, as Figure 24 shows, there is a significant difference between the nested and the CTE templates for some of the queries in group D. For the first two queries, labeled A, which contain two text searches, each of the three templates generates the same execution plan, except for additional projections inserted in the CTE plan due to the **SELECT** clauses of the individual CTEs in the query. These additional operations disappear at the MAL layer.

There are four queries that follow the schema `lemma="müssen" & lemma=`X` & #1 .* #2`, with a varying selectivity of X. For the three queries labeled B_{1-3}, the nested/FROM template outperforms the CTE template. Similarly to the join plan in Figure 18, the second search term is evaluated first, followed by the predicates implementing the indirect precedence operator, and finally by the selections of the first join term. For the query B_1, the evaluation of the indirect precedence operator in this join plan produces an intermediate result containing more than 62,000 entries, which is significantly more than the 99 tuples produced by the indirect precedence operator in the CTE join plan. Indeed, if only one thread is used to evaluate the query, the CTE template is somewhat faster than the nested/FROM template. However, as Figure 25 shows, the join plan generated by the nested templates makes better use of an additional thread than the CTE join plan. As the selectivity of X decreases, this advantage vanishes and the CTE template outperforms the nested/FROM template regardless of the number of threads, e.g., the query B^*. The nested/WHERE template first evaluates the first search term leading to

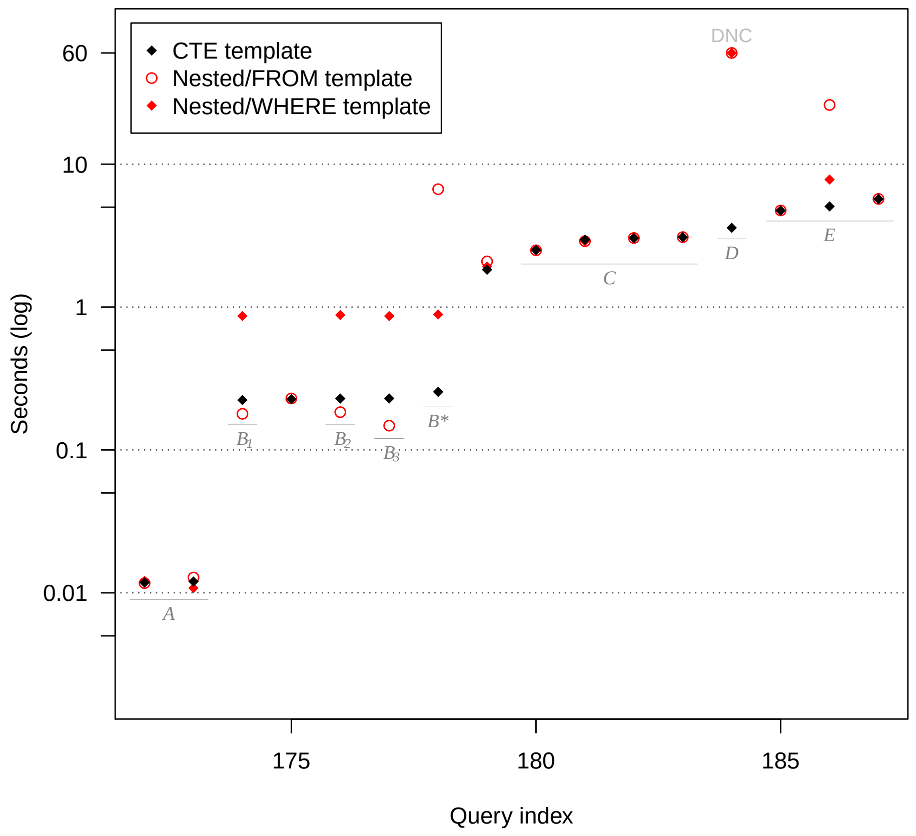


Figure 24: Performance of queries in group D depending on the SQL template.

a constant runtime regardless of the selectivity of the second search term. Similarly, the CTE template pushes both selections down and also has a constant runtime.

The four queries labeled C contain one regular expression which dominates their evaluation time. Similarly, the three queries labeled E contain two regular expressions. The query labeled D contains a negated regular expression search. The SQL query generated by the nested templates does not finish within 60 seconds, whereas the CTE template requires only a little more time than what is needed to evaluate the regular expression by itself.

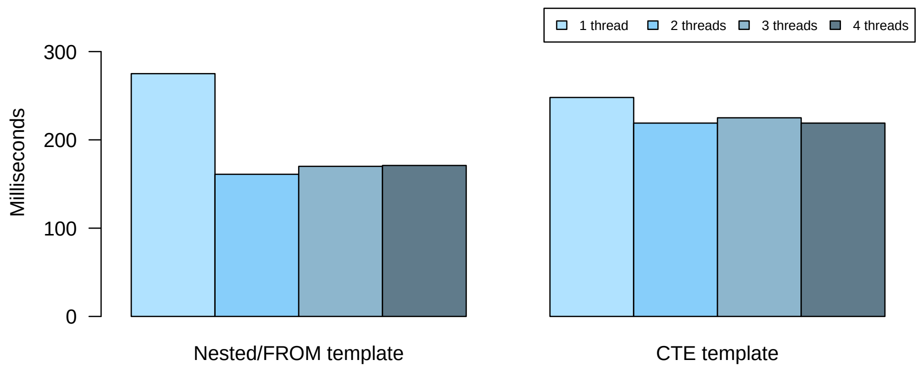


Figure 25: Influence of the number of threads on the runtime of query 174.

Group E

Many of the queries in group E do not complete within 60 seconds using the nested templates. These are marked DNC Figure 26. However, except for the query labeled *A*, the queries in this group indicate a misunderstanding of the Annis query language by the user. They try to limit unbounded indirect precedence operators with punctuation marks or assume that the negated annotation search is fully quantified. Consequently, these queries produce huge result sets as indicated in Table 13.

To evaluate the query *A*, the CTE and the nested templates first compute candidates for each search term before evaluating the linguistic operations referencing them. However, before evaluating the third search term, the nested templates first compute the join of the first and second search term. The CTE template first evaluates each search term, then joins the second with the third, and joins the first search term at the end. Although the second query plan creates smaller intermediate results, as Figure 27 shows, the choice between these two query plans by MonetDB is coincidental.

The queries labeled *B* search for a non-existing annotation value and therefore produce no results. Even though the query could be aborted once it is known that there are no candidates for a search term, both the nested and the CTE templates still evaluate the two regular expression search terms in the query. Thus, all three templates need about 4.8 seconds to evaluate the query. This is twice the time required to evaluate a regular expression *pos* annotation search.

The query labeled *C* contains a direct and an indirect precedence operation. The CTE and nested/WHERE template evaluate the direct precedence operator first and require about 6.9 seconds to finish. The nested/FROM template evaluates the indirect precedence operation first which produces a much bigger intermediate result. Consequently, it requires 53 seconds to finish.

The remaining queries contain two indirect precedence operations. The query labeled *D* produces a

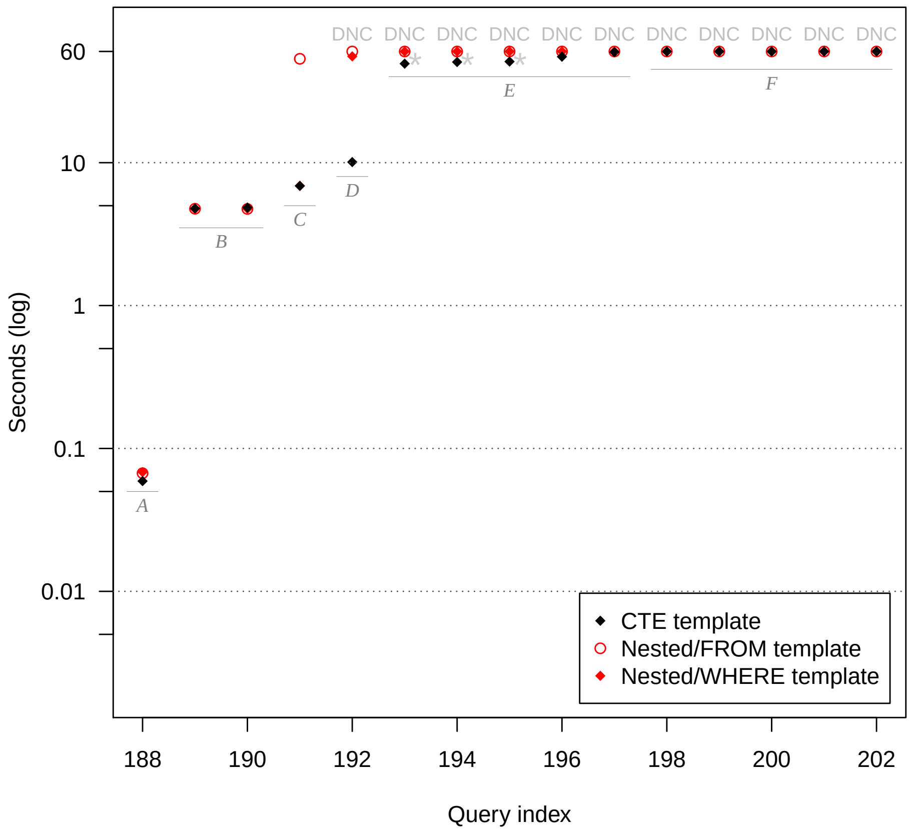
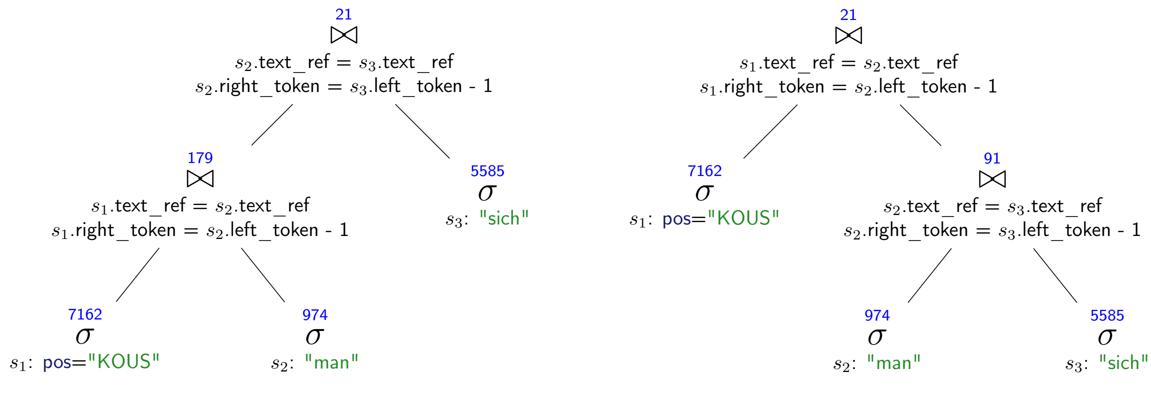


Figure 26: Performance of queries in group E depending on the SQL template.


21
⋈
s_2.text_ref = s_3.text_ref
s_2.right_token = s_3.left_token - 1
179
⋈
s_1.text_ref = s_2.text_ref
s_1.right_token = s_2.left_token - 1
5585
σ
s_3: "sich"
7162
σ
s_1: pos="KOUS"
974
σ
s_2: "man"
21
⋈
s_1.text_ref = s_2.text_ref
s_1.right_token = s_2.left_token - 1
7162
σ
s_1: pos="KOUS"
91
⋈
s_2.text_ref = s_3.text_ref
s_2.right_token = s_3.left_token - 1
974
σ
s_2: "man"
5585
σ
s_3: "sich"

(a) Nested strategies

(b) CTE strategy

Figure 27: Query execution plans for query 188 of group E.

Table 13: Runtime (in seconds) and data written to disk (in MB) for long-running queries in group E.

Query		Time	Results	Disk I/O
192	lemma = "müssen" & pos = /VV.*/ & pos = "$." & #1.*#2 & #2.*#3	10	4934027	< 0.1
193*	pos = /VM.*/ & pos = /VV.*/ & pos = /\$./ & #1.*#2 & #2.*#3	50	53381806	2920
194*	pos = /VM.*/ & pos = /VV.*/ & pos = /\$.*/ & #1.*#2 & #2.*#3	50	53381806	2847
195	pos = /VM.*/ & pos = /VV.*/ & pos = /(\$.|\$,)/ & #1.*#2 & #2.*#3	50	53381806	3026
196	pos = /VM.*/ & pos = /VV.*/ & pos = /\$,/ & #1.*#2 & #2.*#3	55	21195563	8.3
197	pos = /VM.*/ & pos = /VV.*/ & pos = "$." & #1.*#2 & #2.*#3	59	19880160	0.3
198*	lemma = "müssen" & pos != /VV.*/ & pos="$." & #1.*#2 & #2.*#3	614	56465383	3811
199*	lemma = "wollen" & pos != /VV.*/ & pos="$." & #1.*#2 & #2.*#3	243	32066327	1289
200*	lemma="müssen" & pos!="VV.*" & pos="$." & #1 .* #2 & #2 .* #3	591	61399410	4409
201	pos = /VM.*/ & pos != /VV.*/ & pos="$." & #1.*#2 & #2.*#3	*>1 hour*	230726493	
202	pos = /VM.*/ & pos = /VV.*/ & pos = /.*/ & #1.*#2 & #2.*#3	*>1 hour*	383753400	

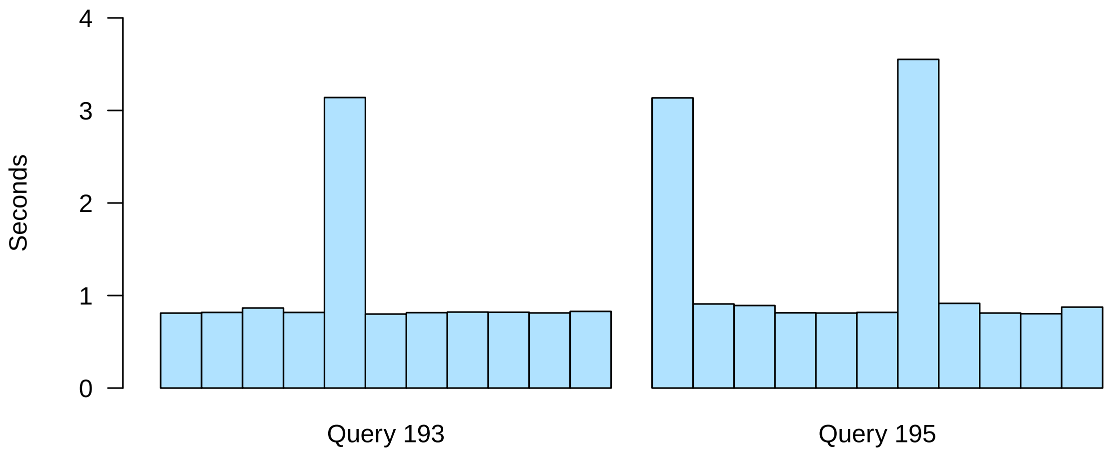


Figure 28: Runtime spikes during consecutive runs of two queries in group E.

relatively small result set containing about 5 million results. The CTE template can evaluate this query in 10 seconds and the nested/WHERE template just barely manages to evaluate the query in under 60 seconds. The CTE strategy is able to evaluate the queries labeled E, which produce between 20 and 50 million results, within 60 seconds. However, the queries labeled F require 4 minutes or longer to evaluate using the CTE template as indicated by Table 13. These queries either contain a negated regular expression search or return more than 100 million results.

During the evaluation of the queries in this group we notice a huge amount of data being written to disk, a phenomenon that other queries do not exhibit. However, no data was ever read. The last column in Table 13 lists the amount of data written to disk as reported by `iostat` averaged over 10 consecutive runs of each query. It takes about 5 hours to test the nine queries in the table; in this time less than 1 MB is read from disk. We suspect that these disk writes are caused by the usage of memory-mapped files for large intermediate results. When these intermediates are freed, i.e., when the files are unlinked, the data is flushed to disk by the operating system. Memory for smaller intermediates is simply allocated using `malloc`. Thus, queries which do not have large intermediate results do not cause disk I/O.

Whereas the runtime of multiple consecutive runs of a query is typically fairly constant, we notice few but very large spikes during the execution of the queries in group E. This phenomenon is illustrated for query 193 and query 195 in Figure 28. Each bar represents an individual run of the query used to compute the mean runtime shown in Figure 26. Query 193 requires on average 49 seconds to evaluate. However, the fifth run requires 188 seconds, almost triple the usual runtime. Similarly, query 195 requires about 51 seconds, but the first and seventh run require 188 and 213 seconds respectively. Notice that there is no such spike during the measurement of disk I/O for query 195. Given the large amount of disk I/O during the execution of these queries, we suspect that these apparently random spikes are caused by interference from other processes, such as writing logfiles to disk. We disregard these outliers for the average runtimes reported in Figure 26 and Table 13. Queries for which outliers are removed are marked with an asterisk.

Group F

The queries in group F best show the advantage of the CTE template. Compared to group E, none of the queries in this group are invalid. The nested templates outperform the CTE template for some of the queries labeled B_{1-3}, but the gains are much smaller than the improvement when the situation is reversed, e.g., the query labeled A. For some queries, the CTE template performs better than the nested templates by an order of magnitude. Even though these queries contain comparatively many search terms and linguistic operations, they finish in well under one second whereas the nested templates sometimes do not even complete in 60 seconds. The two queries labeled D_1 and D_2 contain the Inclusion operator which generally performs slowly in MonetDB. One query, labeled E, includes a *cat* annotation search with a regular expression and another query, labeled F, uses a regular expression text search. The three queries labeled G contain two *pos* annotation searches with a regular expression. Of these, one query does not finish within 60 seconds using the nested templates. The queries labeled C contain a **`root`** unary constraint, which causes a syntax error when evaluated using the nested templates.

6.3 Regular expression searches

Queries containing regular expression search terms are dominated by the evaluation of these search terms. MonetDB requires more than two seconds to evaluate a single regular expression and each additional regular expression increases the query runtime by at least that amount. Some annotation schemes, e.g., the *morph* annotation layer in the TIGER Treebank, encode multiple properties in a single string and therefore require regular expressions to evaluate them. Thus, the optimization of regular expression searches is crucial.

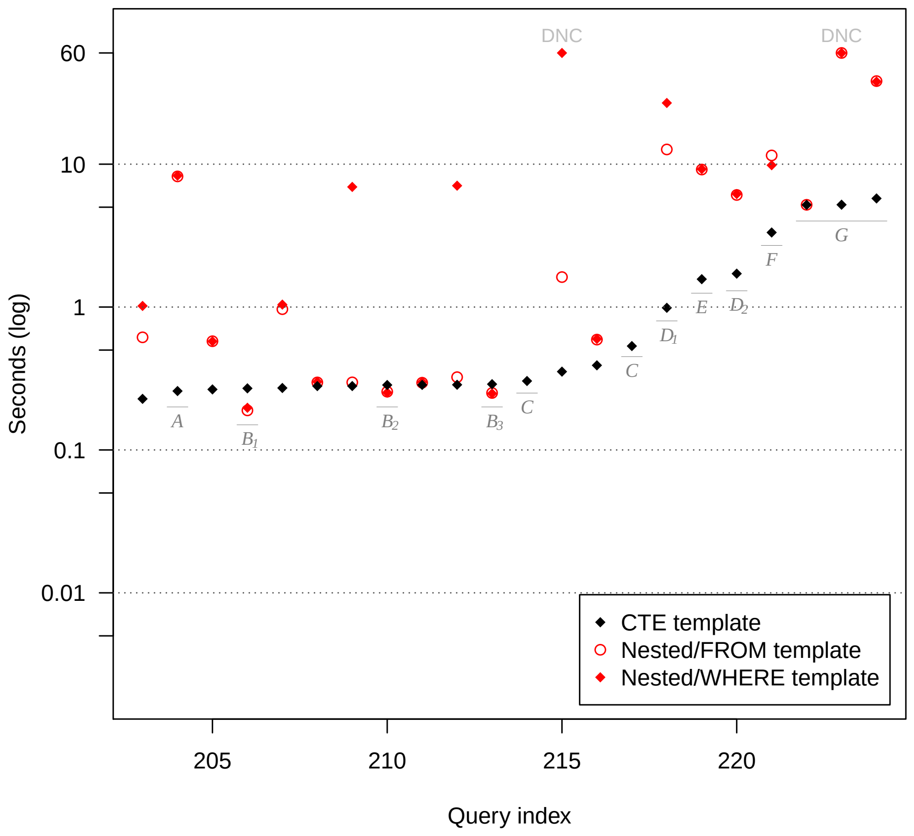


Figure 29: Performance of queries in group F depending on the SQL template.

6.3.1 Implementation of regular expression searches

The poor performance of regular expression searches can be traced back to the evaluation of the `pcre.match` function in the generated MAL plan. Listing 8 shows the MAL plan for a text search containing a regular expression. To generate this and the remaining MAL plans in this section, the MonetDB database is accessed in single-threaded, read-only mode. First the entire `span` BAT is loaded in lines 4 to 7 and a new BAT X_{49} is created in line 8. Lines 9 to 13 contain a loop in which `pcre.match` is called for each `span` value and the result of this evaluation is stored in the BAT X_{49}. This BAT is then restricted to those rows where the evaluation of `pcre.match` succeeded in line 16.

The MAL plan evaluating an exact string match, shown in Listing 9, is fundamentally different. The `span` BAT is restricted to rows matching a given string in a single MAL operation in line 5. MonetDB still needs to scan the entire `span` BAT and create a BAT holding the results. However, as these operations are moved from the MAL layer to a C function, they can be done much faster.

There are three possible approaches to improve the performance of regular expression searches. The first aims to reduce the number of iterations of the loop in Listing 8. This can be achieved by adding additional information to the SQL query. The second option is to remove the loop altogether and match the entire contents of a BAT against a regular expression in a single MAL operation. This approach requires changes to the MonetDB kernel. Finally, both approaches can be combined. Even if the matching loop is moved from the MAL layer to a C function, reducing the input size may still lead to a noticeable performance gain.

```
# A0 contains the regular expression
function user.s0_1(A0:str):void;
    X_2 := sql.mvc();
    X_3:bat[:oid,:str]  := sql.bind(X_2,"annis","node","span",0);
    X_8:bat[:oid,:oid]  := sql.bind_dbat(X_2,"annis","node",1);
    X_10 := bat.reverse(X_8);
    X_11:bat[:oid,:str]  := algebra.kdifference(X_3,X_10);
    X_49 := bat.new(nil:oid,nil:bit);
barrier (X_52,X_53,X_54) := bat.newIterator(X_11);
    X_56 := pcre.match(X_54,A0);
    bat.insert(X_49,X_53,X_56);
    redo (X_52,X_53,X_54) := bat.hasMoreElements(X_11);
exit (X_52,X_53,X_54);
    X_11:bat[:oid,:str]  := nil:bat[:oid,:str];
    X_12:bat[:oid,:bit]  := X_49;
    X_15 := algebra.uselect(X_12,true:bit);
    X_16 := algebra.markT(X_15,0@0:oid);
    X_17 := bat.reverse(X_16);
    X_18:bat[:oid,:int]  := sql.bind(X_2,"annis","node","id",0);
    X_20 := algebra.kdifference(X_18,X_10);
    X_21 := algebra.leftjoin(X_17,X_20);
    (ext48,grp46) := group.done(X_21);
    X_24 := bat.mirror(ext48);
    X_25 := algebra.leftjoin(X_24,X_21);
    X_26 := aggr.count(X_25);
    sql.exportValue(1,"annis.solutions","L1","wrd",64,0,6,X_26,"");
end s0_1;
```

Listing 8: MAL plan generated for a regular expression text search.

```
# A0 contains the search term
function user.s0_1(A0:str):void;
    X_2 := sql.mvc();
    X_3:bat[:oid,:str]  := sql.bind(X_2,"annis","node","span",0);
    X_8 := algebra.uselect(X_3,A0);
    X_9:bat[:oid,:oid]  := sql.bind_dbat(X_2,"annis","node",1);
    X_11 := bat.reverse(X_9);
    X_12 := algebra.kdifference(X_8,X_11);
    X_13 := algebra.markT(X_12,0@0:oid);
    X_14 := bat.reverse(X_13);
    X_15:bat[:oid,:int]  := sql.bind(X_2,"annis","node","id",0);
    X_17 := algebra.leftjoin(X_14,X_15);
    (ext32,grp30) := group.done(X_17);
    X_20 := bat.mirror(ext32);
    X_21 := algebra.leftjoin(X_20,X_17);
    X_22 := aggr.count(X_21);
    sql.exportValue(1,"annis.solutions","L1","wrd",64,0,6,X_22,"");
end s0_1;
```

Listing 9: MAL plan generated for an exact text search.

6.3.2 Minimizing regular expression match loop iterations

Skipping `NULL` values

The runtime of a regular expression search term depends on whether it is a text search or an annotation search, and for the latter, which annotation key is used. This is illustrated by the three search terms in Table 14 selecting either every token or every *pos* or *cat* annotation in the TIGER Treebank. The last column contains the number of tuples in the table that is accessed by the search term. The first search term is evaluated on the `node` table and the other two are evaluated on the `node_annotation` table. The text search and the *pos* search return roughly the same number of results, however the text search takes longer even though the table it scans is shorter. The *cat* and *pos* searches need to scan the same table, but the *cat* search is more than twice as fast as the *pos* search. This speed difference can be traced to the respective selectivity of the annotation key. During the evaluation of a regular expression annotation search, MonetDB will first determine the tuples matching the annotation key and then evaluate the regular expression on the corresponding annotation values. Since there are fewer *cat* than *pos* annotations, the former is faster. Indeed, the speed difference is proportional to the difference in the number of returned results.

In contrast, during the evaluation of a regular expression text search, MonetDB will test every value of the `span` attribute as there is no other predicate to restrict the result. The `span` attribute contains a value only for token spans; for non-token spans it is **`NULL`**. It is the overhead of processing these **`NULL`** values that is causing the performance differences between the text search and the *pos* search in Table 14. Indeed, as Figure 30 shows, the text search requires the same amount of time to produce its result set as the *pos* annotation search. The lightly shaded area at the top of the left bar corresponds to the time spend processing **`NULL`** values in the `span` attribute. Therefore, the performance of a regular expression text search should be improved by introducing a **`NOT NULL`** predicate on `span` to reduce the number of values against which the regular expression is evaluated.

Table 14: Number of results and runtime (in ms) for three regular expression searches.

Search term	Results	Runtime	Table size
`tok=/.*/`	888,563	2,842	1,262,014
`pos=/.*/`	888,578	2,423	3,039,170
`cat=/.*/`	373,436	1,021	3,039,170

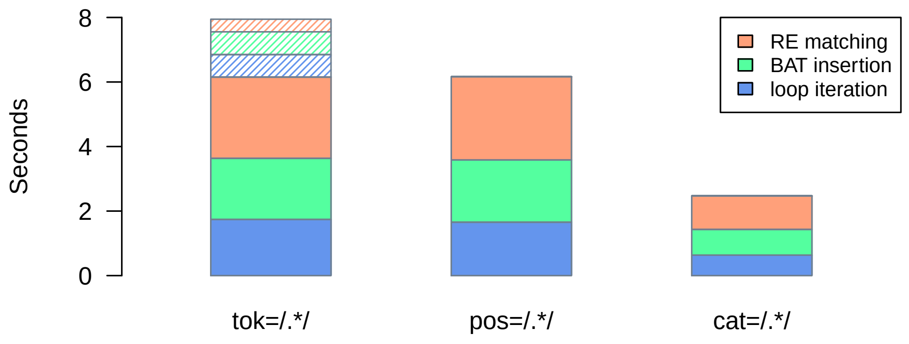


Figure 30: Breakdown of the time spent in the regular expression matching loop.

Upper and lower boundaries for matched values

Every value matched by a regular expression such as `'^VM.*$'` has the same prefix, in this case `'VM'`. Since Annis implicitly anchors regular expressions, we can always find such a fixed prefix, unless the regular expression starts with a character group, e.g., `/[Dd]as/`, an optional parenthesized subexpression, e.g., `/(ge)?kommen/`, the dot special character, e.g., `/.+nd..?/`, with a character followed by an asterisk or question mark, e.g., `/m?ich/`, or if the regular expression consists of multiple alternatives starting from the beginning, e.g., `/Hund|Katze/`. The fixed prefix can be used as a lower bound for the values that have to be matched against the regular expression. Strings that are lexicographically less than the prefix can be skipped. An upper bound can be constructed by replacing the last character of the fixed prefix with its successor. Thus, to match a set of values against the regular expression `'^VM.*$'`, it is sufficient to test the values in the range between `'VM'` (inclusive) and `'VN'` (exclusive).

Replacing regular expressions matching exactly one word

Our test set contains many queries in which a regular expression search term that matches exactly one string, e.g., `pos=/VVFIN/`, is used. Since regular expressions are implicitly anchored by Annis, the `node_annotation.value` attribute is matched against the regular expression `'^VVFIN$'`. This can be replaced with a point predicate, i.e., the search term is internally substituted with the equivalent exact string search `pos="VVFIN"`. This replacement removes the overhead of compiling the regular expression. Furthermore, MonetDB may transparently use a hash index to speed up point predicates on a BAT containing strings.

6.3.3 BAT-aware regular expression matching

MonetDB creates the loop in Listing 8 because it cannot find a faster way to evaluate the predicate **`pcre_match`**`(span, '^...$')` in the SQL query. The signature of the underlying MAL function is `pcre.match(s:str,pat:str):bit`, i.e., the function takes two strings as arguments and returns **`TRUE`** or **`FALSE`**. However, the first argument in the SQL predicate references the attribute `span` of the `node` table. This reference can be interpreted in two ways: As the value of the `span` attribute for a specific tuple in the `node` table or as the entire set of values contained in the `span` attribute. When MonetDB encounters such a call, it will try to use a BAT-aware version of the particular function. A BAT-aware function takes one or more BATs as its arguments, returns a BAT, and its module name is prefixed with *bat*, e.g., `batpcre.match(s:bat[:any_1,:str],pat:str):bat[:any_1,:bit]`. Only if such a function is not available, MonetDB will insert a loop in the MAL plan iterating over the values of a BAT as a fallback.

```
# A0 contains the regular expression
function user.s0_1(A0:str):void;
    X_2 := sql.mvc();
    X_3:bat[:oid,:str]  := sql.bind(X_2,"annis","node","span",0);
    X_8:bat[:oid,:oid]  := sql.bind_dbat(X_2,"annis","node",1);
    X_10 := bat.reverse(X_8);
    X_11 := algebra.kdifference(X_3,X_10);
    X_12:bat[:oid,:bit]  := batpcre.match(X_11,A0);
    X_13 := algebra.uselect(X_12,true:bit);
    X_14 := algebra.markT(X_13,0@0:oid);
    X_15 := bat.reverse(X_14);
    X_16:bat[:oid,:int]  := sql.bind(X_2,"annis","node","id",0);
    X_18 := algebra.kdifference(X_16,X_10);
    X_19 := algebra.leftjoin(X_15,X_18);
    (ext48,grp46) := group.done(X_19);
    X_22 := bat.mirror(ext48);
    X_23 := algebra.leftjoin(X_22,X_19);
    X_24 := aggr.count(X_23);
    sql.exportValue(1,"annis.solutions","L1","wrd",64,0,6,X_24,"");
end s0_1;
```

Listing 10: MAL plan using the BAT-aware version of `pcre.match`.

In order to move the loop from MAL layer to a C function, we have implement a BAT-aware version of `pcre.match`. This function compiles the regular expression only once. It automatically handles **`NULL`** values, i.e., the output BAT only contains entries for those values in the input BAT that are not **`NULL`**. Listing 10 shows a MAL plan for a regular expression text search using the BAT-aware version of `pcre.match`. The loop contained in Listing 8 is replaced with a single call to `batpcre.match` in line 8. The returned BAT is then restricted to those entries which match the regular expression in the next line.

6.3.4 Performance comparison of regular expression searches

Figure 31 shows the runtime of text and annotation searches that contain a regular expression using different techniques to improve their performance. Blue circles represent the time required to evaluate a search term using the default version of `pcre.match` and no optimizations. Red stars depict the runtime of each search term when **`NULL`** values are skipped. Green circles show the evaluation time for the BAT-aware version of `pcre.match`. Blue and green diamonds represent the default and the BAT-aware versions of `pcre.match` for regular expressions with a fixed prefix respectively. Finally, black circles show the evaluation time for queries in which the regular expression is replaced by an exact string match. The chart is sorted according to the query index in appendix B.2.

As expected, the performance of text searches is somewhat improved by the exclusion of **`NULL`** values. However, annotation searches incur a small performance penalty due to the additional predicate. The test is necessary for negated regular expression searches if the default version of `pcre.match` is used because it returns **`FALSE`** if a regular expression is matched against a **`NULL`** value. The BAT-aware version automatically handles **`NULL`** values and removes the corresponding tuples from the output BAT.

Making the `pcre.match` function BAT-aware improves the performance of regular expression searches by

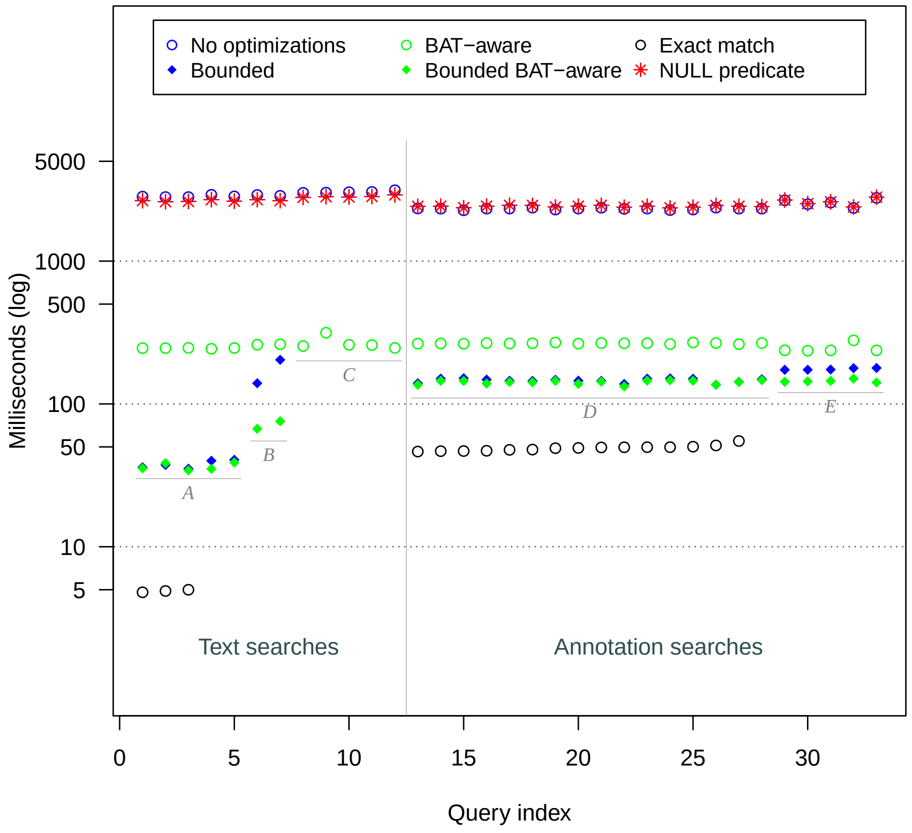


Figure 31: Runtime of regular expression searches using different optimizations.

an order of magnitude. For text searches, this is still 50 times slower than an exact string match. Another improvement by a factor of three to seven is possible if the range of tested values is restricted by a fixed prefix, depending on the number of matched values. For annotation searches, the difference between a BAT-aware regular expression match and an exact string match is only a factor of five. Restricting the range of tested values provides an improvement of factor two.

The performance of the queries in Figure 31 shows that the biggest improvement is gained by restricting the range of tested values. If the fixed prefix is highly selective, e.g., for the queries labeled A and D, there is virtually no difference between the default version of `pcre.match` and the BAT-aware version. Text searches are improved by a factor of more than 70 and annotation searches by a factor of about 15 compared to the default version of `pcre.match` with no other improvements. If a regular expression has no fixed prefix, e.g., the queries labeled C, the BAT-aware version of `pcre.match` provides a performance gain by an order of magnitude compared to the default version. Regular expression searches with a prefix of low selectivity, e.g., queries B and E, benefit the most from a combination of both approaches.

6.4 Binary string searches

Many Annis operations are implemented using string searches on a particular attribute. A text search tests the `node.span` attribute. An annotation search term evaluates the attributes `name` and `value` of the `node_annotation` table. Edge annotations are implemented similarly on the `edge_annotation` table. Finally, if a search term is referenced in a dominance or pointing relation operation, the attributes `type` and `name` of the `component` table are tested. MonetDB uses a linear scan of the corresponding BAT to evaluate these predicates. In general, the entire BAT will be scanned, unless the range has been restricted by another predicate in the SQL query.

The traditional approach to improve the performance of a table or attribute scan is to use B-Tree indexes. However, MonetDB does not support any kind of index. Yet, MonetDB will exploit the fact that a table is sorted on a particular column. In this case, MonetDB will use a binary scan instead of a linear scan to evaluate predicates on this column. Therefore, we expect the performance of text searches to improve by sorting the `node` table on the span attribute. For annotation searches we have to decide between the attributes `name` and `value` of the `node_annotation` table. The selectivity of `value` is much higher and as we have seen in section 5.1, the annotation value in many cases uniquely identifies the annotation name in the TIGER Treebank. We therefore expect a bigger improvement if the `node_annotation` table is sorted on this attribute. We do not test the effect of a sorted `edge_annotation` or `component` table because operations referencing these tables are used much less in our test set.

We use the simple queries containing only one search term from group A to evaluate the influence of a sorted table on query performance. In Figure 32, black circles represent the performance of text searches on an unsorted `node` table and blue diamonds the performance on a table that is sorted on the `span` attribute. The chart is ordered according to the query index in appendix B.3. We do not notice a difference for exact string matches, or regular expression searches that are replaced by an exact string match, which return only a few results, which are labeled A_1. Search terms that return many results, labeled A_2, require longer than search terms returning only a few results on an unsorted table. On the sorted table, this difference vanishes. The most dramatic improvement is experienced by regular expression searches that have a highly selective fixed prefix. The performance of these queries, labeled B_1, is improved by a factor of five on the sorted table. Indeed, they are almost as fast as an exact string match. If the selectivity of the prefix is low, e.g., the queries labeled B_2, the improvement is not as pronounced, but these queries are still more than twice as fast on the sorted table. Finally, the performance of regular expression searches without a fixed prefix, labeled C, is improved by about 15%.

Figure 33 shows the influence of a sorted `node_annotation` table on the performance of annotation searches. Black circles again represent the unsorted `node_annotation` table. Blue and green diamonds show the performance on a table sorted by the annotation name and value respectively. Exact string matches, or regular expression searches that are replaced by an exact string match, show little or no difference, e.g., the queries labeled A_1 for *word*, A_2 for *cat*, and A_3 for *pos* or *lemma* annotation searches. The trivial *lemma* annotation search, labeled B, is *slower* on a sorted table, regardless of the attribute

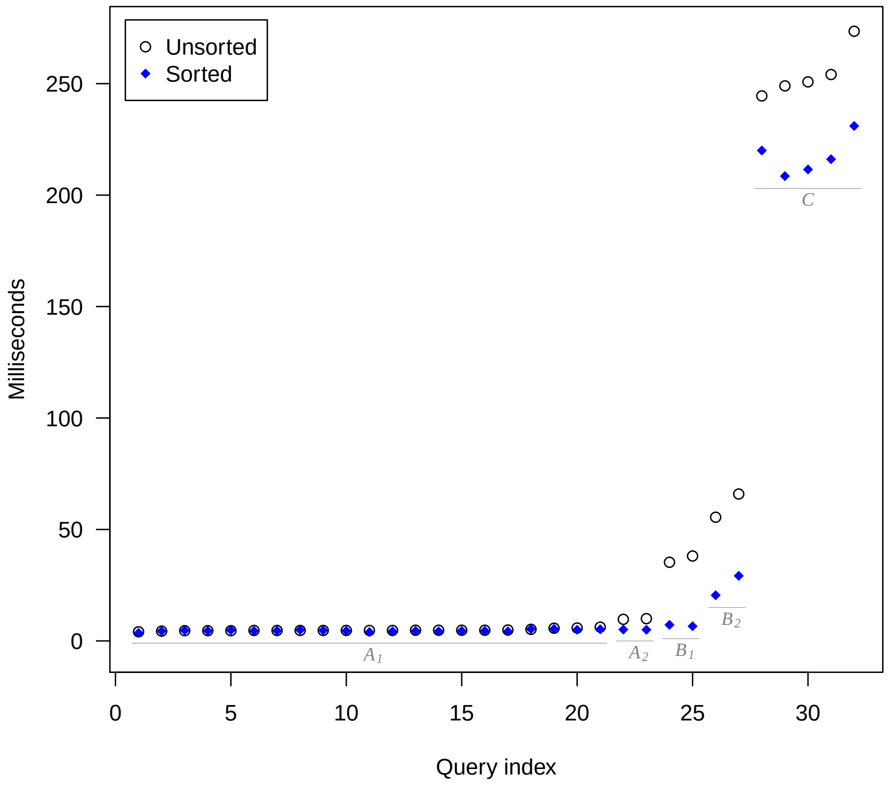


Figure 32: Query performance on a sorted `node` table.

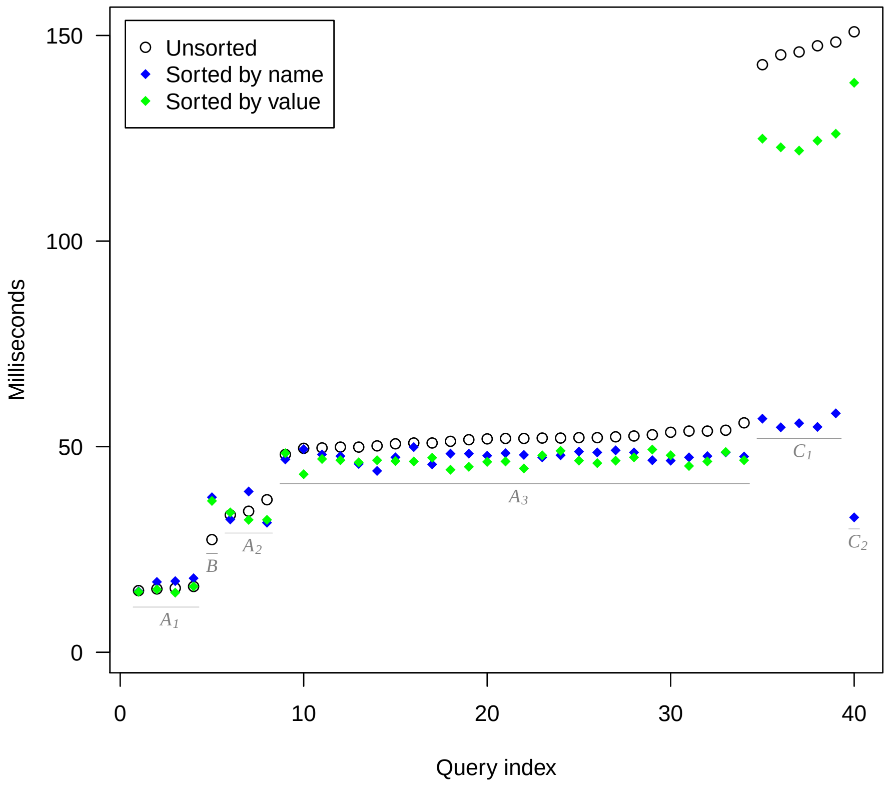


Figure 33: Query performance on a sorted `node_annotation` table.

on which the table is sorted. However, the performance of regular expression searches with a fixed prefix is improved dramatically. For the queries labeled C_1, the selectivity of the prefix is about 0.3% of the entire `node_annotation` table. These regular expression queries are almost as fast an exact string match when evaluated on the `node_annotation` table sorted by the annotation value. The prefix of the query labeled C_2 does not select any tuple in the `node_annotation` table. Surprisingly, this query is even faster than an exact string match.

Contrary to our expectations, annotation searches, at least regular expression searches with a fixed prefix, are faster when the `node_annotation` table is sorted on the annotation name and not the annotation value. In Table 15 we summarize the time spent in different operations during the evaluation of the search term `pos=/VM.*/`. Indeed, sorting the `node_annotation` table on the annotation value leads to a larger reduction in the time required to evaluate the string predicates contained in the query than sorting the table on the annotation name. However, the time required by other operations in the query is also affected by sorting. The dominant operation is a semijoin that filters the `value` BAT to those tuples whose corresponding entry in `name` BAT matches *pos*. This operation is much faster if the table is sorted by the annotation name, leading to an overall faster performance of the query.

Table 15: Time spent (in ms) during the evaluation of a regular expression annotation search.

Operation	**Unsorted table**	**Sorted table** by value	by name
Predicate on `node_annotation.name`	28	9.9	0.1
Predicate on `node_annotation.value`	15	0.1	14
Regular expression matching	4.5	4.2	4.5
Semijoin filtering	**85**	**72**	**31**
Remaining operations	11.5	8.8	10.4
Total	144	95	60

6.5 Deduplication of query solutions

The performance of some queries in group A suggests that the evaluation time of an Annis query is influenced by the number of returned results. For example, the queries 23 and 24, labeled A^* in Figure 20, search for the definite articles *die* and *der* and return about 25,000 results. Both queries require about 20 milliseconds to evaluate, three times longer than query 18 which searches for the definite article *das* and returns about 6,000 results. Similarly, the runtime of *pos* and *lemma* annotation searches, labeled B_3 in Figure 20, as well as *cat* annotation searches, labeled B_2 and B_{2^*}, also increases with the number of returned results. The most striking examples are the trivial searches **`node`** and **`tok`**, labeled C in Figure 20. The **`tok`** search is almost 50 times slower than a text search even though it simply tests the `span` attribute for values that are not **`NULL`** instead of doing a string comparison on the values of the attribute. The **`node`** search is slower still, even though its **`WHERE`** clause is empty and it only needs to count the number of tuples in the `node` table.

This performance difference is not caused by counting the number of results, i.e., by the call to **`count`**`(*)` in the SQL query. MonetDB keeps track of the number of entries in a BAT and can simply look up this value to implement this function call. Instead, the difference is caused by the **`DISTINCT`** operator. Listing 11 shows the MAL program implementing the Annis query **`tok`**. The operations implementing the **`DISTINCT`** operator are highlighted in orange and the operations implementing the **`count`**`(*)` function are highlighted in green. The time spend in each operation is printed in microseconds on the right side of the program. The **`DISTINCT`** operator is responsible for almost 80% of the entire query time whereas computing the count happens almost instantly.

6.5.1 Annis queries requiring explicit deduplication of query solutions

The **DISTINCT** operator is used in the SQL query to remove possible duplicates from a query's result set and to ensure the correct implementation of the *COUNT* query function. If a query is evaluated on the materialized schema, i.e., on PostgreSQL, the **DISTINCT** operator is always required because the materialized `facts` table contains many tuples for each span. However, on the source schema, the **DISTINCT** operator can be omitted in many cases. If the SQL query accesses the `node` table exclusively, i.e., if the Annis query only contains text, **node** or **tok** searches, and coverage and/or precedence operations, then the usage of the `node` table's primary key as the text span identifier makes duplicate results impossible. However, if a text span is contained more than once in an individual component, e.g., because of secondary edges, an Annis query containing dominance or pointing relation operators may lead to duplication of results. Similarly, if a corpus contains multiple annotation layers that have the same name, but prefixed with different namespaces, an annotation search term may also introduce duplicate results. Both conditions can be tested once the corpus has been imported, and SQL queries can be tailored specifically to the peculiarities of a particular corpus.

In the TIGER Treebank, there are no annotation layers with the same name. The **DISTINCT** operator is therefore not necessary for annotation searches. However, it is necessary for queries containing dominance operators, due to secondary edges in the dominance hierarchy of the TIGER Treebank.

6.5.2 Query performance without explicit deduplication

Table 16 lists the number of results and the runtime in milliseconds of some queries from group A with and without the **DISTINCT** operator in the SQL query. The relative and absolute speed improvement is listed as well. Notably, trivial queries benefit if the **DISTINCT** operator is omitted. The unspecific *lemma* annotation search is more than ten times faster and the performance of the **node** search is improved by a factor of 50. Text and annotation searches are a little faster, but the improvement is not as pronounced. The variation of regular expression searches without a fixed prefix is typically higher than the variation

```
  Time   Operation
    15   function user.s0_1():void;
     2       X_1 := sql.mvc();
     8       X_2:bat[:oid,:str] := sql.bind(X_1,"annis","node","span",0);
     4       X_7:bat[:oid,:oid] := sql.bind_dbat(X_1,"annis","node",1);
     5       X_9 := bat.reverse(X_7);
    13       X_10 := algebra.kdifference(X_2,X_9);
 14824       X_11:bat[:oid,:bit] := batcalc.isnil(X_10);
 10582       X_12 := algebra.uselect(X_11,false:bit);
    11       X_13 := algebra.markT(X_12,0@0:oid);
     4       X_14 := bat.reverse(X_13);
    13       X_15:bat[:oid,:int] := sql.bind(X_1,"annis","node","id",0);
    10       X_17 := algebra.kdifference(X_15,X_9);
 16863       X_18 := algebra.leftjoin(X_14,X_17);
126762       (ext46,grp44) := group.done(X_18);
    13       X_21 := bat.mirror(ext46);
 36383       X_22 := algebra.leftjoin(X_21,X_18);
     8       X_23 := aggr.count(X_22);
    13       sql.exportValue(1,"annis.solutions","L1","wrd",64,0,6,X_23,"");
     6   end s0_1;
206127   X_5:void := user.s0_1();
```

Listing 11: Trace of the MAL program for the Annis query **tok**.

of exact string matches. Therefore, an improvement of these searches is only noticeable for queries which return a large number of results.

The influence of the **`DISTINCT`** operator increases with the number of results of the Annis query until it is the dominant factor. This effect is most evident in the performance of the queries in group E. Table 17 lists the runtime in seconds of those queries in group E which return more than one million results. Queries returning hundreds of millions of results, which previously would not finish within an hour, can be evaluated in a few minutes if the **`DISTINCT`** operator is omitted. Indeed, without the **`DISTINCT`** operator, the ratio between the number of results of a query and its runtime is fairly constant. If the **`DISTINCT`** operator is included, the number of results is a much less reliable predictor of the runtime, e.g., query four and eight in Table 17. Furthermore, the occurrences of spikes, signified by an asterisk next to the runtime of a query, is greatly reduced.

Table 16: Runtime (in ms) of queries in group A with and without **`DISTINCT`**.

Annis query	**Results**	**`DISTINCT`**		**Improvement**	
		Yes	No	Relative	Absolute
`lemma`	888,578	291	26	11	264
`tok`	888,578	293	21	14	272
`node`	1,262,014	352	6.9	51	346
`"das"`	6,082	7.1	6	1.2	1.1
`"die"`	24,467	21	10	2.1	11
`"der"`	26,779	22	10	2.2	12
`pos="VVPP"`	17,770	58	52	1.1	6.1
`pos="VVFIN"`	35,628	59	48	1.2	11
`pos="ADJA"`	54,534	64	48	1.3	16
`/.*sich.*/`	7,686	258	257	1	0.2
`/.*und.*/`	22,589	256	254	1	2.7
`/.*s.*/`	212,862	313	274	1.1	39

Table 17: Runtime (in seconds) of queries in group E with and without **`DISTINCT`**.

	Annis query	**Results**	**`DISTINCT`**		**Ratio**
			Yes	No	($\times 10^6$)
1	`lemma = "müssen" & pos = /VV.*/ & pos = "$." & #1.*#2 & #2.*#3`	4,934,027	7.9	2.2	2.2
2	`pos = /VM.*/ & pos = /VV.*/ & pos = "$." & #1.*#2 & #2.*#3`	19,880,160	55	7.3	2.7
3	`pos = /VM.*/ & pos = /VV.*/ & pos = /\$,/ & #1.*#2 & #2.*#3`	21,195,563	49	8	2.7
4	`lemma = "wollen" & pos != /VV.*/ & pos="$." & #1.*#2 & #2.*#3`	32,066,327	237	13	2.5
5	`pos = /VM.*/ & pos = /VV.*/ & pos = /(\$.|\$,)/ & #1.*#2 & #2.*#3`	53,381,806	*45	20	2.6
6	`pos = /VM.*/ & pos = /VV.*/ & pos = /\$.*/ & #1.*#2 & #2.*#3`	53,381,806	44	21	2.5
7	`pos = /VM.*/ & pos = /VV.*/ & pos = /\$./ & #1.*#2 & #2.*#3`	53,381,806	*45	20	2.6
8	`lemma = "müssen" & pos != /VV.*/ & pos="$." & #1.*#2 & #2.*#3`	56,465,383	*614	21	2.6
9	`lemma="müssen" & pos!="VV.*" & pos="$." & #1 .* #2 & #2 .* #3`	61,399,410	591	23	2.7
10	`pos = /VM.*/ & pos != /VV.*/ & pos="$." & #1.*#2 & #2.*#3`	230,726,493	*>1 hour*	107	2.2
11	`pos = /VM.*/ & pos = /VV.*/ & pos = /.*/ & #1.*#2 & #2.*#3`	383,753,400	*>1 hour*	*175	2.2

6.6 Influence of optimization strategies

In order to measure the influence of each optimization strategy, we construct a workload of 10,000 randomly chosen queries from our test set. We replicate the original distribution of the queries in the random workload. As we have seen in section 6.2.3, some queries need more than 60 seconds to evaluate on MonetDB when the corresponding SQL query is generated using the nested/FROM template. Because it is not possible to specify a query timeout in MonetDB, we do not include these queries in the random workload. Specifically, we disregard query 184 of group D, the queries 192 to 202 of group E, and the query 223 of group F. We also do not include the queries 214 and 217 in the workload because the corresponding SQL query created by the nested/FROM or nested/WHERE templates causes a syntax error on MonetDB. Altogether, the generated workload consisted of 209 unique queries.

Figure 34 shows the evaluation times of the entire workload that we obtain by successively activating the previously described optimizations techniques to improve the performance of Annis. The first bar to the left depicts the time required by the initial port of Annis to MonetDB using the nested/FROM template to generate SQL queries. For the remaining tests, the SQL queries are generated using the CTE template. The third bar shows the runtime using the BAT-aware version of `pcre.match` and the fixed prefix to improve the performance of regular expression searches, or by replacing a regular expression with an exact string match. For the fourth test, we additionally remove the **`DISTINCT`** operator from the SQL query if the Annis query does not contain a dominance operation. The last two bars show how the system performs on a sorted table. In both cases the `node` table is sorted on the `span` attribute. The `node_annotation` table is sorted on the `value` attribute for the fifth bar and on the `name` attribute for the last bar on the right.

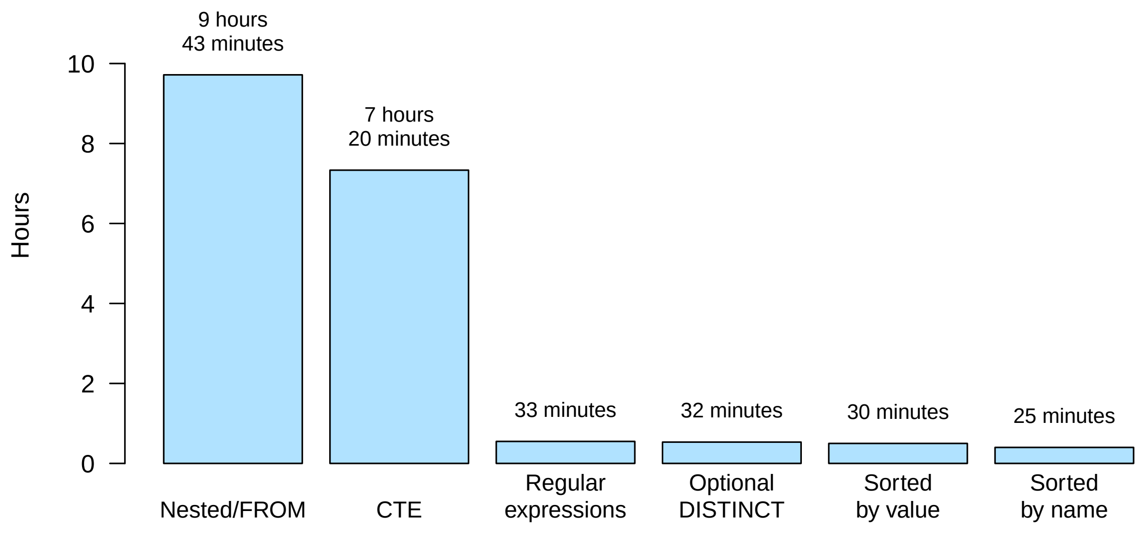


Figure 34: Influence of different optimization strategies.

Using the CTE template instead of the nested/FROM template to generate SQL queries reduces the runtime of the entire workload by about 25%. However, this difference is mostly caused by the ten queries listed in Table 18 which are improved by more than one second by the CTE template. The distribution of the performance gain of the remaining 199 queries is shown in Figure 35. There are ten queries which are faster, up to 85 milliseconds, when the nested/FROM template is used. As we have discussed in section 6.2.3, the query plans generated for these queries using the nested/FROM template evaluate a highly selective search term first and make better use of an additional thread than the query plan that is generated for the CTE template. Another cluster can be found starting at 90 milliseconds. These queries consist of two regular expression searches linked by an Exact Cover operation, i.e., the queries labeled C in Figure 21, or two regular expression and one exact match search linked by two Exact Cover operations, i.e., the queries labeled B in Figure 23.

Table 18: Queries improved by the CTE template by more than one second.

	Query	Nested	CTE	Δ
178	`lemma = "müssen" & lemma = "in" & #1.*#2`	6.7	0.2	6.4
185	`pos = /VM.*/ & pos = /VV.*/ & #1.*#2`	27.8	5.0	22.8
189	`pos = /VM.*/ & pos = /VV.*/ & pos = /\$./ & #1.#2 & #2.*#3`	53.7	6.9	46.8
195	`cat="S" & "umfaßt" & #1 > #2`	8.1	0.3	7.9
201	`pos="NE" & cat="S" & pos="PRELS" & pos="VVFIN" & #2>[func="HD"]#4 & #1$#2 & #3$#4`	1.6	0.4	1.3
204	`/[Jj]e/ & "desto" & #1 $* #2 & morph="Comp" & morph="Comp" & #1 . #3 & #2 . #4`	11.5	3.3	8.1
206	`cat=/(S|VP)/ & lemma="machen" & #1 >edge #2`	8.7	1.6	7.1
207	`cat="S" & pos="VVFIN" & #1 _i_ #2`	12.7	1.0	11.7
208	`cat = "S" & pos = /VM.*/ & pos != /VV.*/ & #1>*#2 & #1>*#3`	40.5	5.8	34.7
209	**`tok`** `& cat="VP" & #1 . #2 &` **`tok`** `& #2 _i_ #3 & #1 $ #3`	6.1	1.7	4.4

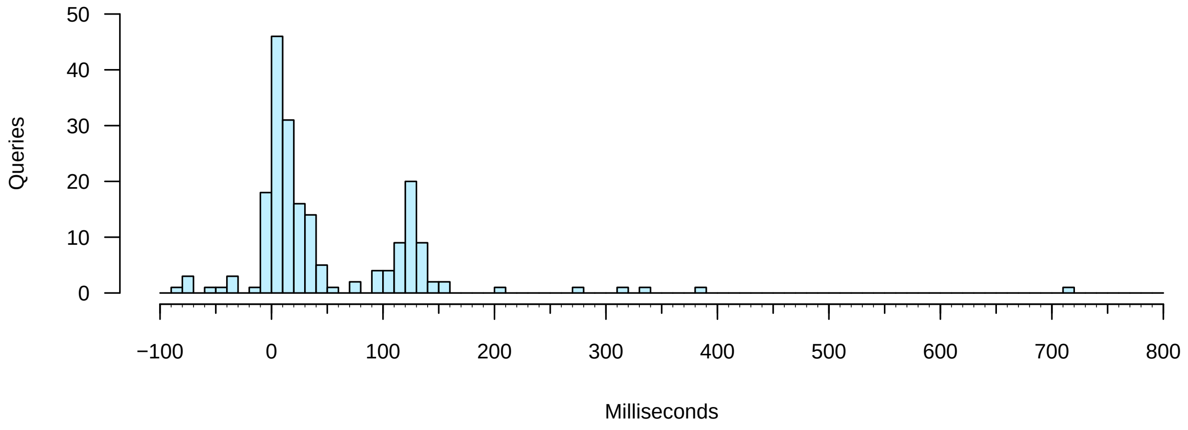


Figure 35: Performance gain of the CTE template.

The most dramatic improvement is achieved by optimizing regular expression searches. This result is not surprising given that 143 queries, more than two thirds, contain one or more regular expression. Figure 36 shows the distribution of the performance gain of these queries. There are four clusters that can be discerned, which correspond to the number of regular expressions found in the query and the type of search term using a regular expression. The first cluster, at 2.3 seconds, consists of queries containing one *pos* or *lemma* annotation search using a regular expression. The next cluster, at 2.7 seconds, contains regular expression text searches. The cluster at 4.5 seconds contains two *pos* or *lemma* regular expression annotation searches. The last cluster, starting at 4.9 seconds, contains a *pos* or *lemma* annotation search and a text search using regular expressions. There are also three outliers visible. The query which shows no improvement uses a regular expression search on the non-existent *token* annotation layer. The query at 1.2 seconds contains a regular expression search on the *cat* layer. Finally, the query at 6.7 seconds contains three regular expression *pos* annotation searches. The runtime of a query containing a regular expression is reduced by 95% on average, or 2.4 seconds for every regular expression found in the query.

The improvement gained by removing the **`DISTINCT`** operator from the SQL query is rather modest, even though this optimization can be applied to 189 queries, or 90% of the test set. However, as we have discussed in section 6.5, the effect of the **`DISTINCT`** operator is only noticeable for queries returning many results. We have removed most of these queries, namely those in group E, from the test set because they do not finish within 60 seconds using the nested/FROM template.

Finally, sorting the `node` and `node_annotation` tables reduces the time required to evaluate the entire workload by another 7% if the `node_annotation` table is sorted on the `value` attribute and 23% if it is sorted on the `name` attribute. There are three queries which are negatively effected by sorting. These are

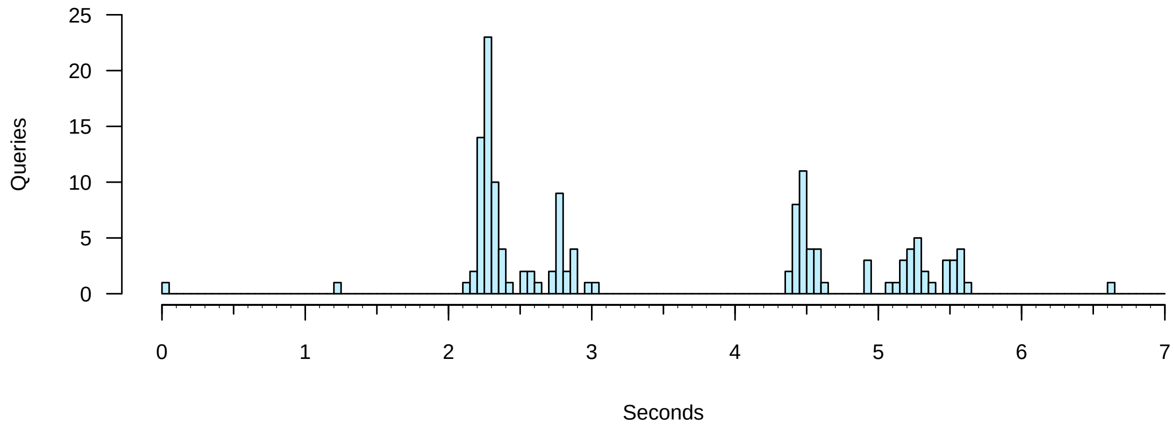


Figure 36: Performance gain of regular expressions searches.

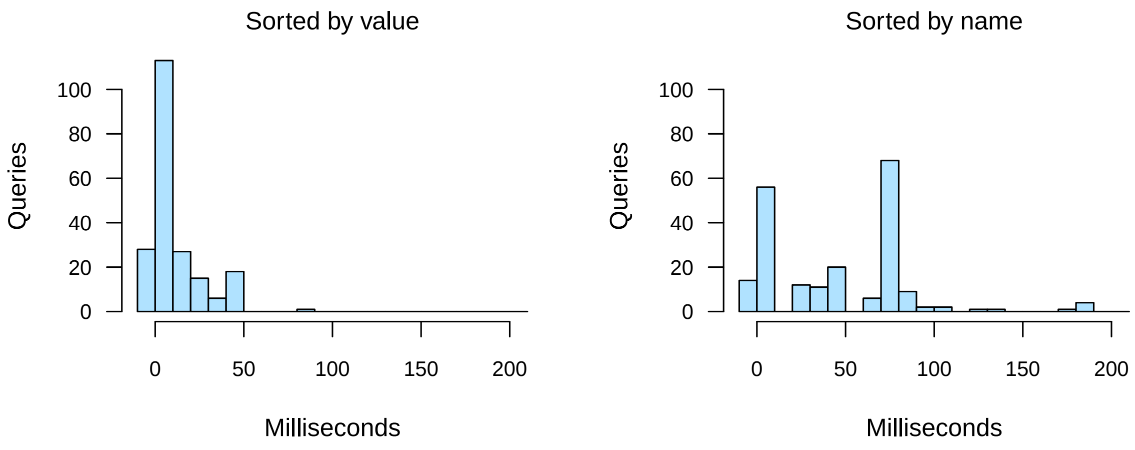


Figure 37: Performance gained by sorting.

listed in Table 19. We have already noticed in section 6.4 that the trivial query `lemma` is slower on the sorted tables. We do not know why these queries are slowed down; if anything their performance should improve. Figure 37 shows how the improvement gained by sorting is distributed. If the `node_annotation` table is sorted on the `value` attribute, the majority of queries either show no or only a very modest improvement. One query is improved by 81 milliseconds, but the mean amount is 11 milliseconds. If the `node_annotation` table is sorted on `name`, the mean improvement is 46 milliseconds and 100 queries are faster by a larger amount. Indeed, many of the queries which gain less than 10 milliseconds have only one search term. These queries are already very fast.

Altogether we were able to achieve an improvement by a factor of 23 from the initial port of Annis to MonetDB. Figure 38 shows the distribution of the runtime of the queries in our test set. Except for four

Table 19: Queries impaired by sorting (ms).

	Query	Unsorted	Sorted		Difference	
			value	name	value	name
37	`lemma`	26	35	37	-8.8	-11
38	`cat="S"`	31	31	40		-9.9
209	`tok & cat="VP" & #1 . #2 & tok & #2 _i_ #3 & #1 $ #3`	1685	1944	1772	-259	-87

queries requiring between 0.8 and 1.8 seconds, every query in our test set finishes in less than 0.5 seconds. Optimizing regular expressions has the greatest effect for two reasons. They are used by many queries and the default implementation of matching regular expressions is particularly inefficient. The main utility of the CTE template is to make the runtime behavior of queries predictable. The runtime of a SQL query generated using the nested templates very often depends on the selectivity of a search term. For SQL queries generated using the CTE template, the runtime mostly depends on the number and type of search terms and on the linguistic operations contained in the query. In particular, except for queries which return a very large result set, the CTE template enables us to evaluate any query in well under 60 seconds. Queries returning many results can be improved by removing the **DISTINCT** operator from the SQL query if this is possible. Sorting the tables on string attributes improves the performance of a wide range of queries by a small amount.

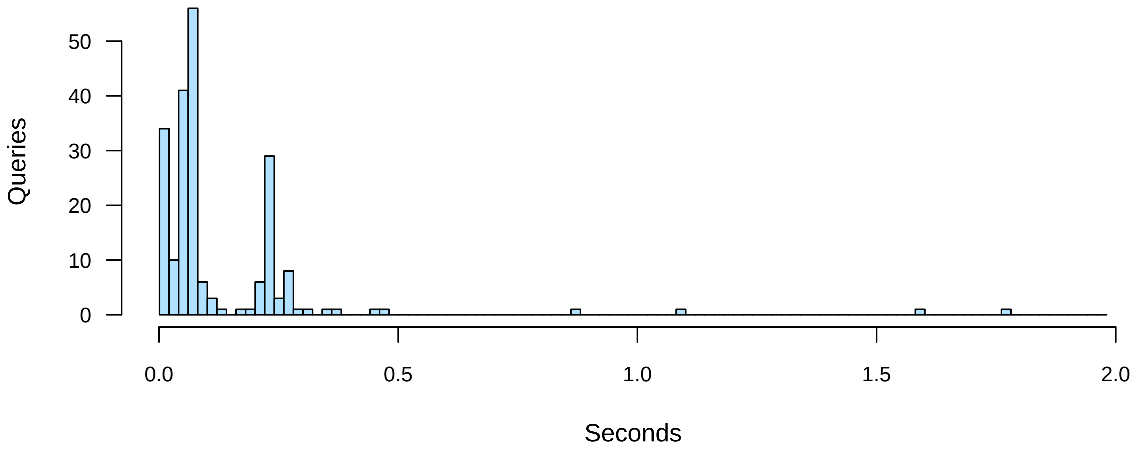


Figure 38: Distribution of query runtime.

6.7 Comparison with PostgreSQL

As a final test, we compare the performance of Annis on MonetDB to the implementation on PostgreSQL. It is worth repeating, that PostgreSQL evaluates the queries on the materialized schema, whereas MonetDB is able to achieve the stated performance on the normalized source schema. This entails an substantial reduction in the disk space required to store the TIGER Treebank by MonetDB.

6.7.1 Disk space consumption

Table 20 shows the disk space required to store the TIGER Treebank in MonetDB and PostgreSQL. MonetDB uses 396 MB, which is less than what is needed by PostgreSQL to store the source tables without indexes. The difference can be explained by unallocated space in the PostgreSQL database page layout that is used to insert new items in the table. Conversely, MonetDB stores insertions and deletions in separate BATs und the original BAT is left unchanged. It can therefore be stored densely. We do not require additional space for indexes in MonetDB. Compared to the materialized schema in PostgreSQL,

Table 20: Disk usage (in MB) of the TIGER Treebank in PostgreSQL and MonetDB.

	Tables	Indexes	Total
PostgreSQL (Source schema)	525	1407	1932
PostgreSQL (Materialized schema)	1201	6707	7908
MonetDB (Source schema)	396		396

the space required by MonetDB to store the TIGER Treebank is reduced by a factor of twenty. However, as we have seen in section 6.5, some queries require a substantial amount of temporary disk space during their evaluation.

6.7.2 Individual query performance

Figure 39 shows the performance of each individual query. Black diamonds indicate the runtime on MonetDB and blue circles the runtime on PostgreSQL. The charts are ordered according to the query index in appendix B.1. The first impression we get from the charts is that there are many queries which are faster on PostgreSQL than they are on MonetDB. Indeed, there are 120 queries, more than half, that perform better on PostgreSQL and only 104 queries that are faster on MonetDB. However, the second fact we notice is that in every group the slowest runtime is from an evaluation of a query on PostgreSQL. As we will see in the next section, the tradeoff between slow and faster queries by MonetDB is much better for overall performance.

It is often possible to deduce the type of query from the comparison between the performance on MonetDB and PostgreSQL. For group A, PostgreSQL is faster for highly selective annotation searches. MonetDB is faster for every other kind of search term, i.e., text searches, annotation searches with a low selectivity, and regular expression searches with either no fixed prefix or a prefix of a low selectivity. Consequently, PostgreSQL is faster than MonetDB for most of the queries in group B because they contain at least one highly selective *lemma* annotation search. MonetDB is faster for queries that contain a regular expression which is not replaced by an exact string match. PostgreSQL performs best, in comparison to MonetDB, for the queries in group C. Again, many of these queries contain a highly selective *lemma* annotation search. The query that gains most by MonetDB contains two regular expression without a fixed prefix. Discounting the invalid queries in group E, MonetDB performs best, compared to PostgreSQL, on the queries in group D. The four queries which are faster on PostgreSQL again contain a highly selective *lemma* annotation search. Indeed, these are the queries B_{1-3} and the query in between them in Figure 24, which also perform better using the nested/FROM template on MonetDB. The remaining queries are faster on MonetDB, sometimes by an order of magnitude. The two queries in group E which are faster on PostgreSQL contain an annotation search that specifies a non-existant annotation value. These are the queries labeled B which we identified in Figure 26. PostgreSQL quickly realizes this fact and aborts the entire query. However, MonetDB evaluates the other two search terms in the queries as we have discussed in section 6.2.3. Most of the remaining queries in group E do not finish within 60 seconds on PostgreSQL. Finally, the queries in group F present a mixed picture. The most apparent feature by which we can distinguish queries that perform better on one system or the other is the number of returned results. Excluding two outliers at 192 and 810, the mean number of results of queries which are faster on PostgreSQL is 8.3. Queries which are faster on MonetDB return more than 9,000 result on average, excluding one outlier that returns more than 200,000 results.

6.7.3 Performance on a random workload

We measure the overall performance of Annis on both systems using two more workloads of 10,000 randomly chosen queries. The first workload contains every query in our test set, including the long-running, invalid queries of group E. To evaluate these 10000 queries, MonetDB requires 5 hours and 31 minutes. PostgreSQL requires 10 hours and 35 minutes. Figure 40 shows how much time is spent processing the queries of a particular group. PostgreSQL spends 85% of the entire time to evaluate the queries in group E. Most of these are aborted after 60 seconds. For MonetDB, this figure rises to 93%. Because we cannot abort queries on MonetDB, much of this time is spent evaluating two queries which require 2 and 3 seconds to finish respectively. Clearly, the invalid queries skew the result considerably. They are overrepresented in the workload and MonetDB is at an disadvantage. Even though it can evaluate these queries faster, it cannot abort them, but has to process them until they are finished.

We therefore construct a second workload of 10,000 queries for which we exclude the invalid queries of group E and run this workload on the server and on the laptop. The outcome is shown in Figure 41.

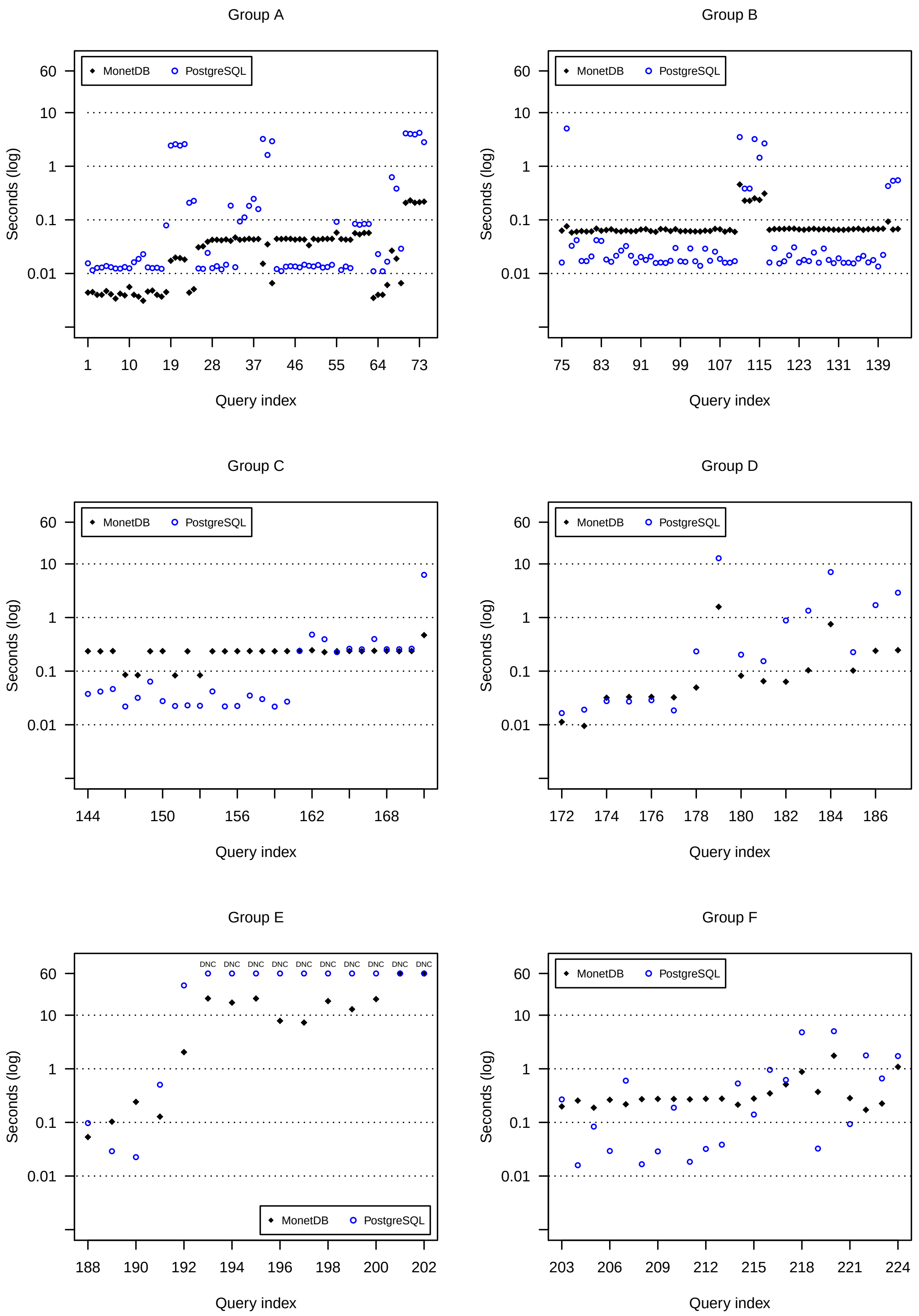


Figure 39: Comparison of individual queries on MonetDB and PostgreSQL.

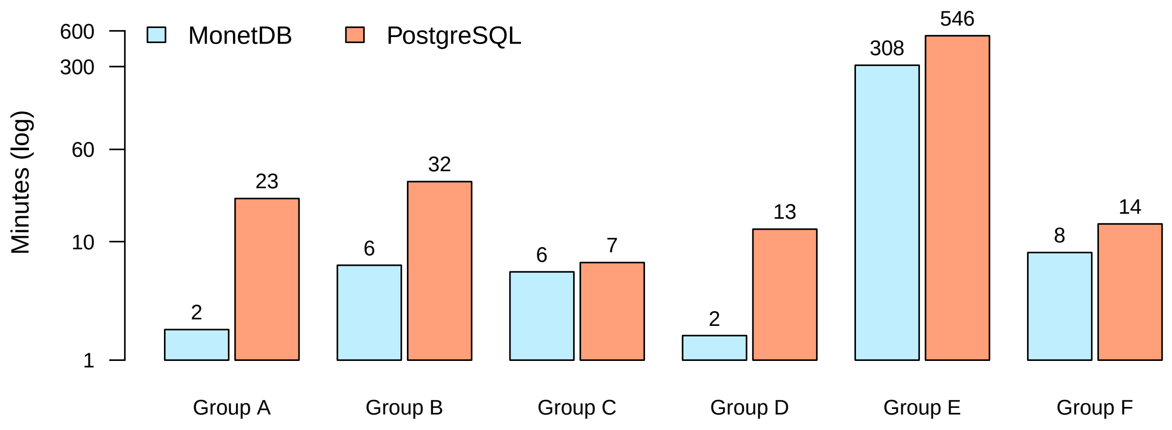


Figure 40: Comparison of a random workload on MonetDB and PostgreSQL by query group.

On the server, MonetDB requires less than a third of the time that PostgreSQL needs to evaluate the workload. On the laptop, the result is even more impressive. MonetDB requires just 15% more time on the laptop than on the server. PostgreSQL on the other hand is more than three times slower. As a result, MonetDB beats PostgreSQL on the laptop by an order of magnitude.

Even though they are more queries that are faster on PostgreSQL in absolute terms, this advantage is outweighed by the overall performance of MonetDB. As an example, consider the queries in group C. Only four queries in this group are clearly faster on MonetDB than on PostgreSQL, one of which by on order of magnitude. Conversely, five queries in this group are faster by on order of magnitude on PostgreSQL, and there are ten other queries where PostgreSQL has an advantage of at least factor five. Yet, as Figure 40 shows, MonetDB beats PostgreSQL overall by a minute in this group. As Figure 38 shows, the three slowest queries on MonetDB require 1.1, 1.6, and 1.8 seconds respectively to evaluate, excluding the invalid queries of group E. In contrast, there are 27 queries that require between one and 13 seconds on PostgreSQL.

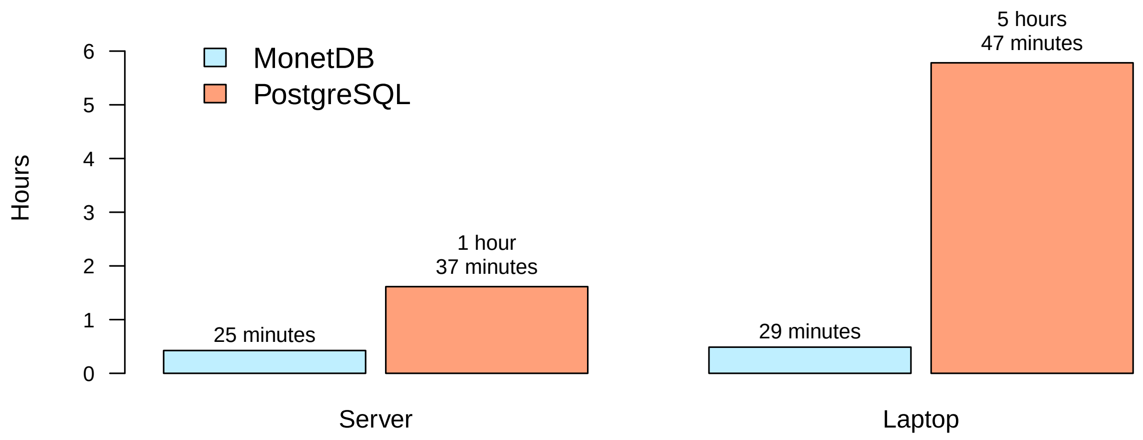


Figure 41: Performance of a random workload on MonetDB and PostgreSQL without invalid queries.

7 Conclusion

The goal of this work was to port the Annis query language to MonetDB and to evaluate Annis queries on the original, normalized source tables so as to retain the flexibility of a normalized database schema. In order to measure the performance of the port and compare it to the implementation on PostgreSQL, we collected 224 unique test queries from an Annis installation at the linguistics department of the Humboldt Universität zu Berlin. These queries were evaluated on the TIGER Treebank.

Our prototypical port supports the entire Annis query language, except for the *ANNOTATE* and *MATRIX* query functions and the selection of subcorpora using metadata. On a server, it is three times faster than the implementation on PostgreSQL. On a consumer laptop, it is only somewhat slower than on the server, whereas the performance of PostgreSQL suffers considerably. Accordingly, the port on MonetDB is ten times faster than the PostgreSQL version on a laptop. This speed improvement is accompanied by a reduction of the space required to store the database on disk by a factor of twenty.

Just a few modifications were necessary to adapt the SQL code generated by Annis to MonetDB. In fact, we only had to change the evaluation of boolean predicates and expose a MAL function to the SQL layer in order to implement regular expression searches. However, the performance of this minimal port was greatly unsatisfying. Even simple queries would require a comparatively long time to finish because of the creation and processing of very large intermediate results during their evaluation. The performance of the minimal port was also very unpredictable. Structurally similar queries would yield widely diverging runtimes depending on the selectivity of the search term that by chance was evaluated first. Furthermore, the evaluation of regular expression searches was very inefficient. Each value of an attribute was tested individually in a loop in the MAL program which involved a lot of overhead.

We used four strategies to improve the performance of Annis on MonetDB. First, we modified the generated SQL code so that MonetDB generates a query execution plan that greatly reduces the size of intermediates. Second, we optimized regular expression searches by processing the tested attribute in a C function and limiting the range of tested values. Third, we improved the performance of string searches by sorting. Fourth, we removed an unnecessary deduplication step in order to improve the performance of queries that return a large number of results.

Originally, an Annis query function used a nested subquery in the **`FROM`** clause to compute the solutions of an Annis query. For these queries, MonetDB produced a join plan which evaluated a binary linguistic constraint by joining multiple copies of the entire `node` table instead of restricting it first to those tuples selected by a search term. We modified the SQL code to compute candidates for each search term in an individual common table expression in a **`WITH`** clause. Binary operators were evaluated in a separate common table expression. We called this approach to the generation of SQL queries the CTE template. It effectively pushed down the computation of search term matches in the query execution plan, thereby reducing the size of intermediate results considerably. This change decreased the time required to evaluate a random workload by 25%. However, the improvement was not uniform across the queries in our test set. Many of them only gained a little more than 100 milliseconds or were not improved at all. Yet, there were also ten queries which were many seconds faster when the CTE template was used, and two queries which previously did not finish within a minute could be evaluated in less than a second.

Another advantage of the CTE template was an increased predictability of the performance of a query. The influence of the selectivity of a query's search terms, and more importantly their order, was greatly reduced. Using a nested subquery, the performance of an Annis query would often have depended on the selectivity of the search term that was evaluated first, a factor which we found very hard to predict, let alone control. Indeed, we found the decisions made by MonetDB, in which order to evaluate the predicates of a SQL query, extremely opaque. There were a few queries, namely those containing a highly selective search term in an indirect precedence operation, that were slowed down by the CTE template, even though the size of their intermediate results was reduced. For these queries, MonetDB used a second thread more effectively when a nested subquery was used. However, as the selectivity of the search term decreased, and the size of intermediate results increased, the time required to process these intermediates dominated the execution time. In this case, the CTE template was faster because it keeps the size of intermediates small. Using the CTE template, the runtime of a query largely depended on the number and type of its search terms, the binary linguistic operations used to join these search terms, and on the number of returned results. Evidently, the latter factor is correlated to the selectivity of the query's search terms, but for queries which do not return a large result set, it was not the deciding factor.

The evaluation of regular expression searches was a major performance bottleneck. MonetDB created a loop in the MAL program to test every value of an attribute individually. Because Annis implicitly anchors regular expressions at the start and at the end, we could, for many queries in our test set, completely side-step the costly evaluation of regular expressions, by replacing them with an equivalent exact string match. However, this was only possible if the implicitly anchored regular expression matched exactly one word. We improved the performance of genuine regular expressions by making the function evaluating them BAT-aware, i.e., moving the loop from the MAL program into a C function. This modification to the MonetDB kernel resulted in a performance gain of an order of magnitude. Regular expressions starting with a fixed prefix could further be improved by constructing a lower and upper bound from the prefix and restricting the evaluation of the attribute to values within this range. This provided another performance gain of a factor of two to seven depending on the type and selectivity of the search term. Optimizing regular expression searches had the greatest effect on the overall performance for three reasons. First, two thirds of the queries in our test set contained at least one regular expression and benefited from their optimization. Second, many of them could be replaced with an exact string match, which for text searches was 500 times faster. Third, we were able to greatly increase the performance of the remaining, genuine regular expressions, up to a factor of 70 for text searches.

Sorting the `node` table on the `span` attribute and the `node_annotation` table on the `name` attribute reduced the time to evaluate a random workload by 23%. About two thirds of the queries in our test set benefited from sorting the tables on a string attribute. Roughly half of them were improved by about 80 milliseconds. The most dramatic performance gain was experienced by regular expression searches that have a highly selective fixed prefix. These queries were almost as fast as an exact string match on a sorted table. We had expected that sorting the `node_annotation` table on the `value` attribute should result in a bigger improvement than sorting it on the `name` attribute, because the former has a higher selectivity and in most cases uniquely identifies the latter. However, the attribute on which the `node_annotation` table was sorted not only influenced the time required to evaluate string predicates, but also affected other operations in the query. As a result, regular expression annotation searches with a fixed prefix were faster if the table was sorted by the annotation name rather than the annotation value.

Finally, some of the queries returned a very large number of results, ranging from tens to hundreds of million matches. We suspected that the user did not fully understand the Annis data model when they formulated these kind of invalid queries. For queries returning many results, the evaluation of the **`DISTINCT`** operator, i.e., the explicit deduplication of the result set, was the dominating factor. Because we evaluated Annis on the normalized source schema, we were able to avoid this explicit deduplication step in many cases. By removing the **`DISTINCT`** operator, we reduced the time required to evaluate these long-running queries dramatically, from many minutes or hours to a few seconds or minutes. Indeed, without the **`DISTINCT`** operator, the runtime was proportional to the size of the result set. However, most queries were unaffected by this improvement because the number of results they return was quite small. An exception were the trivial queries **`node`** and **`tok`**, as well as annotation searches which do not specify a value, such as `lemma`.

Altogether, we improved the performance of the initial port by a factor of 23 for our test set. Every query in our test set could be evaluated in less than two seconds on MonetDB, excluding the previously mentioned invalid queries. Indeed, there were only six queries that require more than 0.5 seconds to evaluate. Those returning up to 60 million results could be evaluated in less than 25 seconds, and queries returning up to 380 million results in less than three minutes.

Our implementation of Annis on MonetDB compared favorably to Annis running on PostgreSQL. On a server with 48 GB of main memory, MonetDB could evaluate a random workload of 10,000 queries in 25 minutes. PostgreSQL was more than three times slower and required 97 minutes for the same workload. On a laptop with 4 GB of main memory, MonetDB was only 15% slower, requiring a little less than half an hour to evaluate the workload. PostgreSQL required almost six hours, which is a difference by an order of magnitude. Furthermore, a number of queries with many results, which required substantially longer than a minute on PostgreSQL, finished within a few seconds on MonetDB. Because we evaluated queries directly on the source tables in MonetDB and did not use indexes, the size of the database on disk was greatly reduced. Whereas PostgreSQL needed almost 8 GB to store the TIGER Treebank, MonetDB required less than 400 MB.

The port was instructive insofar as it exposed a few bugs in the MonetDB query processor and a long-standing bug in PostgreSQL related to the processing of regular expressions.

We did not implement any of the corpus management or selection facilities found in Annis and assumed that the database contained only one corpus. The obvious next step is to implement these features as well as the $ANNOTATE$ and $MATRIX$ query functions, in order to use the implementation on MonetDB as a back-end of the Annis web interface. We expect that the remaining features can be implemented similarly to $COUNT$ using common table expressions in a **`WITH`** clause.

Even though the TIGER Treebank consists of comparatively many tokens, it is not the most complex corpus in Annis. For example, it does not contain pointing relations. Furthermore, the queries in our test set were not overly complex, having a maximum number of four search terms. We primarily chose the TIGER Treebank as a test set because it allowed us to collect queries from actual users of the system and construct a realistic test workload. It would be instructive to test the limits of Annis on MonetDB using a larger or more complex corpus and queries consisting of more search terms.

We would also like to test our assumption that the evaluation of Annis queries on the normalized source tables improves the flexibility of the data model. After all, this was one of the main motivations of the port to MonetDB. The addition of multiple precedence orders can provide an interesting test case. This feature would allow Annis to support multiple token orders, e.g., for error-annotated learner corpora, or different levels of precedence, e.g., syllables as subtokens. Especially the former feature is often requested by Annis users. Multiple precedence levels could be implemented by extracting the attributes `left_token` and `right_token` from the `node` table and storing them in an additional `precedence` table along with a new attribute that discriminates the precedence level.

However, compared to other linguistic query languages, the major feature lacking in Annis is meaningful support of negation. As we have seen in the test queries, the current implementation of existential value negation is confusing to users.

To summarize, we have shown that a main-memory, column-oriented database system is a good foundation to implement a linguistic query language. We have achieved a considerable gain in performance compared to a traditional database system and at the same time reduced the resource requirements in terms of disk space and main memory size.

A Additional Annis features

In this appendix we list Annis features which we did not implement on MonetDB and describe the necessary steps to port them.

A.1 The $ANNOTATE$ query function

The $ANNOTATE(q, C, left, right[, offset, limit])$ query function returns for each solution S to a query q in a set of corpora C an annotation graph fragment consisting of every text span overlapping a span in S with $left$ tokens as left context and $right$ tokens as right context. Formally, if $A = (V, E)$ is an annotation graph, the annotation graph fragment for a solution S with context $left$ and $right$ consists of the node set

$$V' = \bigcup_{s \in S} \{v \in V : v \text{ overlaps a token from the interval } [min_s - left, max_s + right]\}$$

and the edge set

$$E' = \{(v, w) \in E : v, w \in V'\} .$$

The optional parameters $offset$ and $limit$ are used to paginate the output of the $ANNOTATE$ query function. The solutions are sorted, the first $offset$ solutions are skipped and the annotation graph fragments for $limit$ solutions are returned.

The $ANNOTATE$ function can be implemented by wrapping the SQL query computing the solutions S to q as a subquery and then retrieving the overlapping tokens of the annotation graph fragment over S in the outer query. This requires a modification of the **SELECT** clause of the inner query as shown in Listing 12. In addition to the node IDs, the attributes `text_ref`, `left_token`, and `right_token` are selected in order to compute overlapping text spans. The outer query is depicted in Listing 13. The PostgreSQL-specific **ARRAY** constructor is used to gather the `node.id` attributes of the spans in a query solution into an array in order to create a key, grouping all the spans of a retrieved annotation graph fragment (line 1). The solutions are sorted and only those specified by the parameters $offset$ and $limit$ are returned by the wrapped query (line 6). If no pagination is requested this line can be skipped. The construction of the annotation graph requires the tables `node`, `rank`, `component`, `node_annotation`, and `edge_annotation` (lines 2 and 14). Some additional information may be retrieved to provide context for the web frontend (line 3). Finally, the output of the query is ordered by the key identifying an annotation graph fragment and the pre-order value to ease the reconstruction of the annotation graph in the application (line 21).

The result set returned by this query can be transformed into an annotation graph using an algorithm that is similar to the gXDF reconstruction of a DDDquery result described in [Vit04]. A simple walk through the result set represents a pre-order traversal of the graph we want to reconstruct. We keep track of the nodes and edges already visited as well as their annotations to skip through the result set if possible. Once we encounter a new key, we know that the current annotation graph fragment is complete and start a new one.

```
SELECT DISTINCT
  node1.id AS id1, node1.text_ref AS text1,
    node1.left_token - left AS min1, node1.right_token + right AS max1,
  ...,
  nodeN.id AS idN, nodeN.text_ref AS textN,
    nodeN.left_token - left AS minN, nodeN.right_token + right AS maxN
```

Listing 12: **SELECT** clause for the inner query of the $ANNOTATE$ query function.

```
SELECT DISTINCT ARRAY[solutions.id1, ..., solutions.idN] AS key,
                facts.*,
                corpus.path_name
FROM (
       SQL subquery to compute the solutions to q with a modified SELECT clause
       ORDER BY id1, ..., idN OFFSET offset LIMIT limit
     ) AS solutions,
     (
       node
       JOIN rank ON (rank.node_ref = node.id)
       JOIN component ON (rank.component_ref = component.id)
       LEFT JOIN node_annotation ON (node_annotation.node_ref = node.id)
       LEFT JOIN edge_annotation ON (edge_annotation.rank_ref = rank.pre)
     ) AS facts,
     corpus
WHERE ( facts.text_ref = matches.text1 AND
        facts.left_token <= solutions.max1 AND facts.right_token >= solutions.min1 )
OR    ...
OR    ( facts.text_ref = matches.textN AND
        facts.left_token <= solutions.maxN AND facts.right_token >= solutions.minN )
ORDER BY key, facts.pre
```

Listing 13: SQL query template for the $ANNOTATE$ query function.

The $ANNOTATE$ query function extends the inner query with **LIMIT**, **OFFSET**, and **ORDER BY** clauses to paginate the output. Unfortunately, MonetDB does not support these clauses for nested queries, view definitions, or common table expressions. It is therefore necessary to temporarily materialize the result of the nested query. The materialized result can then be used in place of the nested subquery in the outer SQL query of the $ANNOTATE$ function. The temporary result has to be deleted once the query has finished. To ensure this, we wrap the whole process in a transaction that is rolled back after completion. The procedure is illustrated in Listing 14.

Furthermore, the $ANNOTATE$ query function uses the tuple of node IDs that represents a match to an AQL query as a key to group the nodes of the annotation graph of the match. This key is encoded as an array of node IDs, e.g., `ARRAY[solution.id1,..., solution.idN]`. This syntax is PostgreSQL-specific. In MonetDB, we simply return the node ID columns in the outer query and create the key in application code. The array functionality of PostgreSQL is also used for the `corpus.path_name` attribute. For each document of a corpus this attribute stores the names of the ancestor documents up to the corpus root. When exported from PostgreSQL, the textual representation of this attribute is {document name, parent name, ..., corpus name}. This value is stored as a **VARCHAR** attribute in MonetDB and converted to an array in application code.

```
BEGIN;
CREATE TABLE solutions AS
  SQL query to compute solutions with a modified SELECT clause
  ORDER BY id1,..., idN OFFSET offset LIMIT limit
WITH DATA;
SQL query for the ANNOTATE query function
accessing the previously materialized solutions table
ROLLBACK;
```

Listing 14: SQL query template for the $ANNOTATE$ query function on MonetDB.

A.2 The $MATRIX$ query function

The $MATRIX(q, C)$ query function retrieves span and corpus annotations for each span that is returned as part of a solution to the query q in a set of corpora C. The result is returned as a table in an ARFF file. This table contains a row for each solution S to the query. For each span in a solution there exists a set of columns containing the internal node ID of the span, the text covered by it, as well as one column for any annotation key that is used for a span at this position in any solution. These columns are grouped by the span position in the solution tuple. Additionally, there is one column for any corpus annotation key that is attached to any document containing solutions to q. Cells in the table may be empty if there is no span at this position in the solution, i.e., if the query consists of multiple alternatives or if the span, or the document containing the span, does not have an annotation specified by the column. An example of such a table is provided in Listing 15 for the query `cat="S" & #1:tokenarity=1,6 & pos="VVFIN" & #1 > #2` evaluated on the PCC corpus. Spans at the first position have only one annotation (line 5), spans at the second position have three annotations (lines 8 through 10). There are no corpus annotations. The table is constructed with the help of the SQL query shown in Listing 16. It wraps the SQL query computing the solutions to q as a subquery and returns one row for each span that is part of a solution to q (line 15). The PostgreSQL-specific **`ARRAY`** constructor is used to group spans belonging to one solution (line 1). Annotations belonging to a span are aggregated using the PostgreSQL-specific aggregate function **`array_agg`** in order to collapse multiple rows for one span into one row (line 4). The same is done for corpus annotations.

Three changes are necessary to adapt the $MATRIX$ query function to MonetDB: The aggregation of node and corpus annotations must be removed from the SQL query and recreated in the Java application code. Instead of constructing a key array, the node IDs of a solution can be returned in multiple columns. The call to **`substr`** has to be changed to **`substring`**.

```
1   @relation name
2
3   @attribute #1_id string
4   @attribute #1_span string
5   @attribute #1_tiger:cat string
6   @attribute #2_id string
7   @attribute #2_span string
8   @attribute #2_token_merged:lemma string
9   @attribute #2_token_merged:morph string
10  @attribute #2_token_merged:pos string
11
12  @data
13
14  '357','Wunder gibt es immer wieder','S','231','gibt','geben','3.Sg.Pres.Ind','VVFIN'
15  '949','obwohl sie ihnen gegenüber sitzen','S','766','sitzen','sitzen','3.Pl.Pres.Ind','VVFIN'
16  '928','was Jugendliche wollen und brauchen','S','751','brauchen','brauchen','3.Pl.Pres.Ind','VVFIN'
17  '981','Die glänzten diesmal noch mit Abwesenheit','S','513','glänzten','glänzen','3.Pl.Past.Ind','VVFIN'
18  '390','wie sie die Chance verwerten','S','66','verwerten','verwerten','3.Pl.Pres.Ind','VVFIN'
```

Listing 15: ARFF file constructed from the output of the $MATRIX$ query function.

A.3 Corpus selection and metadata

A user can select multiple root corpora on which to perform an Annis query q. Thus, the SQL query computing the solutions to q shown in Listing 2 needs to be restricted to those spans found in the corpora specified by the user and their child documents. A SQL query that computes the solutions found in a set of k root corpora is shown in Listing 17. The number of documents in a corpus can be considerably large, e.g., the TIGER corpus contains nearly 2000 documents. Testing the `node.corpus_ref` attribute against that many values incurs an unnecessary overhead. Therefore the `node` table also contains the `toplevel_corpus` attribute which for each span points to the root corpus containing the document to which the span belongs. This attribute can be compared against the list of root corpora

```
SELECT ARRAY[solutions.id1, ..., solutions.idN] AS key,
       node.id AS id,
       min(substr(text.text, node.left + 1, node.right - node.left)) AS span,
       array_agg(DISTINCT coalesce(node_annotation.namespace || ':', '')
           || node_annotation.name || ':' || node_annotation.value) AS annotations,
       array_agg(DISTINCT coalesce(corpus_annotation.namespace || ':', '')
           || corpus_annotation.name || ':' || corpus_annotation.value) AS metadata
FROM   (
         SQL subquery to compute the solutions to q
       ) AS solutions,
       node
       JOIN text ON (text.id = node.text_ref)
       LEFT JOIN node_annotation ON (node_annotation.node_ref = node.id)
       LEFT JOIN corpus_annotation ON (corpus_annotation.corpus_ref = node.corpus_ref)
WHERE  node.id = solutions.id1 OR ... OR node.id = solutions.idN
GROUP BY key, facts.id, span
```

Listing 16: SQL query template for the $MATRIX$ query function.

specified by the user (lines 9 through 11). Additionally, corpus names provided by the frontend need to be mapped to corpus primary keys. This can be performed in a common table expression in a **WITH** clause (lines 1 through 3). If the SQL query is used as a subquery to compute the result of a query function, the **WITH** clause needs to be moved to the outer query.

A user may also restrict the search to specifically annotated documents by listing meta annotations in an Annis query. A SQL query that only searches the child documents of a set of k root corpora that match a set of l corpus annotations is shown in Listing 18. In this case, the `node.corpus_ref` attribute is compared against the list of matching documents (lines 21 through 23). A test on `node.toplevel_corpus` is superfluous but may be added for performance reasons. The list of documents below the given root corpora that match the specified meta data is computed in a common table expression in a **WITH** clause (lines 1 through 15).

The PostgreSQL implementation of Annis uses horizontal partitioning to store the data of one corpus in a dedicated `facts` table. Only the parent `facts` table is referenced in the generated SQL queries. PostgreSQL transparently evaluates the query on the appropriate child table using the `toplevel_corpus` attribute as a discriminator. MonetDB does not support horizontal partitioning. In order to support multiple corpora in MonetDB, one option is to store the data of every corpus in one set of tables and rely

```
 1  WITH corpora AS ( SELECT id
 2                    FROM   corpus
 3                    WHERE  name IN (corpus_1, ..., corpus_k) )
 4  SELECT DISTINCT node1.id AS id1, ..., nodeN.id AS idN
 5  FROM   corpora,
 6         node AS node1 JOIN additional tables required to evaluate the first search term
 7         ...
 8         node AS nodeN JOIN additional tables required to evaluate the n-th search term
 9  WHERE  node1.toplevel_corpus IN (corpora.id)
10  AND    ...
11  AND    nodeN.toplevel_corpus IN (corpora.id)
12  AND    additional predicates to evaluate the query
```

Listing 17: SQL query template to compute the solutions of an Annis query on a set of corpora.

```
WITH documents AS ( SELECT child.id
                    FROM   corpus AS root,
                           corpus AS child
                           JOIN corpus_annotation AS annotation1
                                ON (corpus.id = annotation1.corpus_ref)
                           ...
                           JOIN corpus_annotation AS annotationL
                                ON (corpus.id = annotationL.corpus_ref)
                    WHERE  root.name IN (corpus_1, ..., corpus_k)
                    AND    child.pre BETWEEN root.pre AND root.post
                    AND    annotation1.name = 'key'_1
                    AND    annotation1.value = 'value'_1
                    AND    ...
                    AND    annotationL.name = 'key'_L
                    AND    annotationL.value = 'value'_L )
SELECT DISTINCT node1.id AS id1, ..., nodeN.id AS idN
FROM   corpora,
       node AS node1 JOIN additional tables required to evaluate the first search term
       ...
       node AS nodeN JOIN additional tables required to evaluate the n-th search term
WHERE  node1.corpus_ref IN (documents.id)
AND    ...
AND    nodeN.corpus_ref IN (documents.id)
AND    additional predicates to evaluate the query
```

Listing 18: SQL query template to compute the solutions of an Annis query with meta data.

on the `toplevel` attribute and MonetDB's foreign key indexes to restrict the data. Another option is to store the data of a corpus in an individual set of tables and reference them directly in the SQL query. Alternatively, a view or common table expression over the union of the required corpus tables could be constructed before the query is evaluated.

A.4 Database administration

The Annis system includes a corpus administration utility that is used to initialize a PostgreSQL database for Annis and to import corpora into the Annis database or delete them from it. This utility creates a Java property file that is used to configure a JDBC connection data in Annis. An example of this file for MonetDB is shown in Listing 19. Specifically, the fully qualified class name of the MonetDB JDBC driver is nl.cwi.monetdb.jdbc.MonetDriver and the URL scheme for a MonetDB database is jdbc:monetdb:.

```
datasource.driver=nl.cwi.monetdb.jdbc.MonetDriver
datasource.url=jdbc:monetdb://localhost:50000/annis
datasource.username=annis
datasource.password=annis
```

Listing 19: JDBC connection properties for MonetDB.

B Test data

In this appendix we list the query runtime data from which we generated the charts in section 6.

B.1 Query groups

The data in the following tables was used to generate the charts in section 6.2.3 and section 6.7.2.

Runtime (in ms) of queries in group A.

Query		Results	FROM	WHERE	CTE	MonetDB	PostgreSQL
1	`tok="gleich"`	108	4.6	4.6	4.2	4.4	16
2	`tok="bewerten"`	11	4.9	4.6	4.3	4.5	12
3	`tok="befassen"`	14	4.7	4.6	4.4	4	13
4	`tok="bedeuten"`	31	4.7	4.6	4.5	4	13
5	`tok=".+nd.?.?"`	0	4.5	4.3	4.6	4.7	14
6	`tok="beginnen"`	34	4.6	4.7	4.6	4.1	13
7	`tok="begreifen"`	9	4.7	4.7	4.6	3.4	12
8	`tok="befürchten"`	38	4.6	4.5	4.7	4.2	12
9	`"jedes"`	59	5.3	5.2	4.8	3.9	13
10	`"jede"`	78	5.2	5.2	4.9	5.6	12
11	`"alles"`	210	5.4	5.3	5	4	16
12	`"was"`	502	5.5	5.3	5	3.7	19
13	`"man"`	974	5.6	5.5	5.2	3.1	23
14	`"Rias"`	2	5.7	5.4	5.2	4.6	13
15	`"Appell"`	17	5.6	5.6	5.4	4.8	13
16	`tok="berichten"`	25	4.8	4.7	5.5	4	13
17	`"6"`	24	6.3	5.9	6.2	3.7	12
18	`"das"`	6082	7.5	7.2	7.2	4.5	79
19	`word="schlafend..?"`	0	16	16	16	17	2432
20	`word="laufend?.?"`	0	16	16	17	20	2578
21	`word="laufend..?"`	0	16	16	17	19	2429
22	`word=".+nd.?.?"`	0	17	15	17	18	2590
23	`"die"`	24467	20	21	20	4.4	208
24	`"der"`	26779	22	22	22	5.1	226
25	`cat="T"`	0	32	35	34	31	12
26	`cat="TP"`	0	36	36	34	32	12
27	`lemma = "müssen"`	1880	54	53	50	39	24
28	`pos="VVIFIN"`	0	50	56	50	42	13
29	`lemma="fragen"`	98	52	52	52	42	14
30	`pos="ADJ."`	0	53	51	52	42	12
31	`pos="VVIMP"`	162	51	53	53	43	15
32	`cat="S"`	72346	57	58	54	41	184
33	`pos="ADJ"`	0	53	51	54	47	13
34	`pos="CARD"`	15939	59	58	57	42	93
35	`pos="VVPP"`	17770	60	56	59	43	111

Runtime (in ms) of queries in group A (continued).

Query		Results	FROM	WHERE	CTE	MonetDB	PostgreSQL
36	pos="VVFIN"	35628	62	61	62	44	182
37	pos= "ADJA"	54534	67	70	68	43	246
38	lemma="--"	121701	84	84	84	44	159
39	**tok**	888578	302	295	295	15	3230
40	lemma	888578	302	311	304	35	1624
41	**node**	1262014	376	379	387	6.6	2926
42	lemma=/brennt/	0	2288	2286	2291	44	12
43	lemma=/brannt/	0	2298	2296	2299	44	11
44	lemma = /brennen/	10	2334	2329	2323	44	13
45	lemma=/besetzt/	33	2330	2328	2328	44	14
46	lemma=/waschen/	8	2336	2330	2335	43	14
47	lemma=/operiert/	0	2428	2364	2356	44	13
48	lemma = /gebrannt/	1	2372	2364	2356	43	15
49	lemma=/brannt?/	0	2351	2338	2356	34	14
50	lemma=/begonnen/	17	2357	2371	2358	44	14
51	lemma=/besuchen/	63	2376	2377	2360	42	14
52	lemma=/gebissen/	0	2420	2357	2361	44	13
53	lemma=/gesunken/	8	2378	2379	2364	44	13
54	lemma=/gelitten/	1	2363	2367	2364	44	14
55	pos = /VM.*/	9325	2367	2377	2365	58	92
56	lemma=/besuchend/	1	2392	2404	2387	44	12
57	lemma=/gewachsen/	19	2420	2411	2391	43	14
58	lemma=/operieren/	16	2410	2398	2401	43	13
59	pos = /VM.*␣\$./	0	2517	2526	2524	56	84
60	pos = /VM.*.*\$./	0	2577	2568	2597	54	81
61	pos = /VM.*␣.*␣\$./	0	2682	2678	2679	57	84
62	pos = /VM.*VV.*.*\$./	0	2780	2775	2786	57	84
63	**tok**=/brannte/	4	2829	2826	2819	3.5	11
64	**tok** = /müssen/	497	2820	2821	2820	4	23
65	**tok**=/gebrannt/	0	2848	2842	2842	4	11
66	**tok** = /bekomm../	88	2860	2850	2855	6.1	17
67	/de.*/	51189	2871	2879	2879	27	625
68	/der.*/	27468	2926	2923	2920	19	383
69	/kann.*/	886	2930	2930	2932	6.6	29
70	**tok**=/.+nd..?/	11920	3024	3021	3010	208	4106
71	**tok**=/.*s.*/	212862	3044	3031	3041	231	4017
72	**tok**=/.*und.*/	22589	3053	3044	3046	209	3914
73	**tok**=/.*sich.*/	7686	3075	3060	3064	213	4206
74	/[Kk]ann.*/	905	3167	3156	3139	219	2812

Runtime (in ms) of queries in group B.

Query		Results	FROM	WHERE	CTE	MonetDB	PostgreSQL
75	`pos="VVPP" & lemma= "gekommen" & #1␣=␣#2`	0	162	185	150	63	16
76	`pos="ADJA" & token=/.+nd..?/ & #1␣=␣#2`	0	224	224	157	76	5066
77	`pos="VVIZU" & lemma=/kommen/ & #2 ␣=␣ #1`	0	2413	2408	2409	58	33
78	`pos="VVINF" & lemma=/kommen/ & #2 ␣=␣ #1`	110	2437	2437	2413	60	42
79	`pos="VVPP" & lemma=/leiden/ & #2 ␣=␣ #1`	2	2426	2437	2418	62	17
80	`pos="VVINF" & lemma=/leiden/ & #2 ␣=␣ #1`	9	2436	2436	2422	61	17
81	`pos="VVINF" & lemma=/sinken/ & #2 ␣=␣ #1`	23	2432	2432	2422	61	21
82	`pos="VVFIN" & lemma=/kommen/ & #2 ␣=␣ #1`	695	2446	2437	2427	69	42
83	`pos="VVPP" & lemma=/kommen/ & #2 ␣=␣ #1`	99	2435	2432	2430	63	40
84	`pos="VVFIN" & lemma=/leiden/ & #2 ␣=␣ #1`	30	2454	2458	2431	65	18
85	`pos="VVFIN" & lemma=/sinken/ & #2 ␣=␣ #1`	61	2451	2450	2433	67	16
86	`pos="VVPP" & lemma=/sinken/ & #2 ␣=␣ #1`	35	2426	2437	2438	62	21
87	`pos="VVINF" & lemma=/führen/ & #2 ␣=␣ #1`	96	2456	2456	2441	61	27
88	`pos="VVPP" & lemma=/führen/ & #2 ␣=␣ #1`	94	2452	2454	2446	63	32
89	`pos="VVINF" & lemma=/wachsen/ & #2 ␣=␣ #1`	25	2469	2464	2455	61	21
90	`pos="VVINF" & lemma=/beißen/ & #2 ␣=␣ #1`	1	2468	2471	2456	62	16
91	`pos="VVFIN" & lemma=/wachsen/ & #2 ␣=␣ #1`	79	2498	2486	2458	66	20
92	`pos="VVFIN" & lemma=/waschen/ & #2 ␣=␣ #1`	3	2488	2488	2458	67	18
93	`pos="VVPP" & lemma=/wachsen/ & #2 ␣=␣ #1`	17	2458	2453	2461	62	21
94	`pos="VVPP" & lemma=/beißen/ & #2 ␣=␣ #1`	0	2456	2465	2461	60	16
95	`pos="VVFIN" & lemma=/beißen/ & #2 ␣=␣ #1`	3	2484	2481	2461	68	16
96	`pos="VVFIN" & lemma=/brennen/ & #2 ␣=␣ #1`	8	2486	2490	2462	67	16
97	`pos="VVINF" & lemma=/waschen/ & #2 ␣=␣ #1`	2	2465	2468	2462	62	17
98	`pos="VVFIN" & lemma=/führen/ & #2 ␣=␣ #1`	216	2467	2484	2462	67	30
99	`pos="VVINF" & lemma=/brennen/ & #2 ␣=␣ #1`	1	2465	2477	2465	61	17
100	`pos="VVPP" & lemma=/brennen/ & #2 ␣=␣ #1`	0	2471	2464	2467	62	16
101	`pos="VVINF" & lemma=/beginnen/ & #2 ␣=␣ #1`	21	2495	2500	2483	61	29
102	`pos="VVINF" & lemma=/besetzen/ & #2 ␣=␣ #1`	6	2505	2498	2485	61	17
103	`pos="VVPP" & lemma=/waschsen/ & #2 ␣=␣ #1`	0	2487	2475	2494	61	14
104	`pos="VVPP" & lemma=/beginnen/ & #2 ␣=␣ #1`	80	2486	2493	2496	63	29
105	`pos="VVPP" & lemma=/besetzen/ & #2 ␣=␣ #1`	25	2487	2492	2501	62	17
106	`pos="VVFIN" & lemma=/beginnen/ & #2 ␣=␣ #1`	154	2516	2515	2504	68	26
107	`pos="VVFIN" & lemma=/besetzen/ & #2 ␣=␣ #1`	8	2522	2508	2505	67	19
108	`pos="VVINF" & lemma=/operieren/ & #2 ␣=␣ #1`	5	2531	2535	2522	60	16

Runtime (in ms) of queries in group B (continued).

Query		Results	FROM	WHERE	CTE	MonetDB	PostgreSQL
109	`pos="VVFIN" & lemma=/operieren/ & #2 _=_ #1`	6	2553	2549	2533	65	16
110	`pos="VVPP" & lemma=/operieren/ & #2 _=_ #1`	5	2530	2531	2541	60	17
111	`lemma="--" & pos!=/\$.*/ & #1 _=_ #2`	2225	3015	3002	2683	457	3505
112	`pos="VVPP" & lemma=/(ge)?kommen/ & #2 _=_ #1`	99	2804	2818	2812	229	383
113	`pos="VVPP" & lemma=/(ge)?brennen/ & #2 _=_ #1`	0	2852	2853	2842	228	384
114	`pos="ADJA" & tok=/.nd..?/ & #1_=_#2`	0	2910	2899	2900	252	3212
115	`pos= "ADJA" & tok= /.+nd..?/ & #1_=_#2`	3677	3080	3077	3078	236	1448
116	`lemma=/.+[^aeiouäöü]chen/ & pos="NN" & #1 _=_ #2`	224	3281	3267	3185	310	2667
117	`lemma=/beehren/ & pos=/VVFIN/ & #1_=_#2`	0	4698	4640	4607	65	16
118	`lemma=/betonen/ & pos=/VVFIN/ & #1_=_#2`	157	4712	4652	4608	68	30
119	`lemma=/behagen/ & pos=/VVFIN/ & #1_=_#2`	1	4708	4646	4609	68	15
120	`lemma=/befassen/ & pos=/VVFIN/ & #1_=_#2`	14	4746	4679	4632	68	17
121	`lemma=/belasten/ & pos=/VVFIN/ & #1_=_#2`	18	4739	4681	4635	69	22
122	`lemma=/beginnen/ & pos=/VVFIN/ & #1_=_#2`	154	4738	4695	4636	69	31
123	`lemma=/beweinen/ & pos=/VVFIN/ & #1_=_#2`	1	4740	4699	4639	66	16
124	`lemma=/bewerten/ & pos=/VVFIN/ & #1_=_#2`	15	4740	4701	4639	65	18
125	`lemma=/begegnen/ & pos=/VVFIN/ & #1_=_#2`	18	4746	4691	4639	67	17
126	`lemma=/bedeuten/ & pos=/VVFIN/ & #1_=_#2`	115	4741	4695	4651	68	25
127	`lemma=/besolden/ & pos=/VVFIN/ & #1_=_#2`	0	4736	4663	4654	66	16
128	`lemma=/berichten/ & pos=/VVFIN/ & #1_=_#2`	238	4774	4725	4667	67	29
129	`lemma=/begreifen/ & pos=/VVFIN/ & #1_=_#2`	17	4770	4714	4672	67	18
130	`lemma=/bespielen/ & pos=/VVFIN/ & #1_=_#2`	0	4775	4705	4677	66	16
131	`lemma=/beziffern/ & pos=/VVFIN/ & #1_=_#2`	26	4774	4715	4687	66	19
132	`lemma=/bemuttern/ & pos=/VVFIN/ & #1_=_#2`	0	4763	4719	4697	65	16
133	`lemma=/begradigen/ & pos=/VVFIN/ & #1_=_#2`	0	4795	4729	4698	66	16
134	`lemma=/berechnen/ & pos=/VVFIN/ & #1_=_#2`	5	4775	4735	4706	67	15
135	`lemma=/beschenken/ & pos=/VVFIN/ & #1_=_#2`	1	4811	4742	4712	69	19
136	`lemma=/befürchten/ & pos=/VVFIN/ & #1_=_#2`	58	4833	4790	4725	65	21
137	`lemma=/bevormunden/ & pos=/VVFIN/ & #1_=_#2`	0	4840	4784	4736	67	16
138	`lemma=/beschichten/ & pos=/VVFIN/ & #1_=_#2`	0	4828	4789	4737	68	18
139	`lemma=/begünstigen/ & pos=/VVFIN/ & #1_=_#2`	7	4868	4805	4780	67	14
140	`lemma=/beschäftigen/ & pos=/VVFIN/ & #1_=_#2`	80	4908	4856	4788	68	22
141	`lemma=/be.*/ & pos=/VVFIN.*/ & #1_=_#2`	3275	4957	4933	4848	93	428
142	`tok=/be.*/ & pos=/VVFIN/ & #1_=_#2`	3250	5156	5163	5146	66	536
143	`tok=/be.+/ & pos=/VVFIN/ & #1_=_#2`	3250	5155	5154	5160	68	549

Runtime (in ms) of queries in group C.

Query		Results	FROM	WHERE	CTE	MonetDB	PostgreSQL
144	`pos="ADJA" & tok=/.+nd..?/ & #1_=_#2 & lemma="kommen" & #3 _=_ #1`	1	3201	3289	3285	236	38
145	`pos="ADJA" & tok=/.+nd..?/ & #1_=_#2 & lemma=/kommen/ & #3 _=_ #1`	1	5528	5525	5424	234	42
146	`pos="ADJA" & tok=/.+nd..?/ & #1_=_#2 & lemma=/kommend/ & #3 _=_ #1`	219	5562	5564	5459	237	46
147	`pos="ADJA" & tok=/be.+t..?/ & #1_=_#2 & lemma=/besetzen/ & #3 _=_ #1`	0	5597	5595	5486	85	22
148	`pos="ADJA" & tok=/ge.+en..?/ & #1_=_#2 & lemma=/führen/ & #3 _=_ #1`	0	5569	5576	5487	84	32
149	`pos="ADJA" & tok=/.+nd..?/ & #1_=_#2 & lemma=/kommend?/ & #3 _=_ #1`	220	5626	5614	5516	235	64
150	`pos="ADJA" & tok=/.+nd..?/ & #1_=_#2 & lemma=/sinkend?/ & #3 _=_ #1`	33	5630	5616	5517	236	28
151	`pos="ADJA" & tok=/be.+en..?/ & #1_=_#2 & lemma=/besetzen/ & #3 _=_ #1`	0	5610	5610	5519	83	22
152	`pos="ADJA" & tok=/.+nd..?/ & #1_=_#2 & lemma=/leidend?/ & #3 _=_ #1`	4	5616	5616	5520	234	23
153	`pos="ADJA" & tok=/ge.+en..?/ & #1_=_#2 & lemma=/besetzen/ & #3 _=_ #1`	0	5614	5612	5525	84	23
154	`pos="ADJA" & tok=/.+nd..?/ & #1_=_#2 & lemma=/führend?/ & #3 _=_ #1`	93	5632	5649	5540	235	42
155	`pos="ADJA" & tok=/.+nd..?/ & #1_=_#2 & lemma=/beißend?/ & #3 _=_ #1`	4	5646	5649	5543	235	22
156	`pos="ADJA" & tok=/.+nd..?/ & #1_=_#2 & lemma=/brennend?/ & #3 _=_ #1`	12	5659	5638	5549	236	22
157	`pos="ADJA" & tok=/.+nd..?/ & #1_=_#2 & lemma=/wachsend?/ & #3 _=_ #1`	69	5651	5647	5550	237	35
158	`pos="ADJA" & tok=/.+nd..?/ & #1_=_#2 & lemma=/beginnend?/ & #3 _=_ #1`	17	5686	5677	5582	235	30
159	`pos="ADJA" & tok=/.+nd..?/ & #1_=_#2 & lemma=/besetzend?/ & #3 _=_ #1`	2	5684	5674	5583	235	22
160	`pos="ADJA" & tok=/.+nd..?/ & #1_=_#2 & lemma=/operierend?/ & #3 _=_ #1`	7	5720	5707	5629	236	27
161	`os="ADJA" & tok=/ge.+en..?/ & #1_=_#2 & lemma=/(ge)?wachsen/ & #3 _=_ #1`	0	5817	5816	5740	240	238
162	`pos="VVPP" & tok=/ge.+en/ & #1_=_#2 & lemma=/(ge)?kommen/ & #3 _=_ #1`	99	5849	5846	5752	245	480
163	`pos="ADJA" & tok=/ge.+t..?/ & #1_=_#2 & lemma=/(ge)?brannt/ & #3 _=_ #1`	1	5875	5869	5785	226	394
164	`pos="VVPP" & tok=/ge.+en..?/ & #1_=_#2 & lemma=/(ge)?kommen/ & #3 _=_ #1`	0	5918	5912	5839	236	226
165	`pos="ADJA" & tok=/ge.+en..?/ & #1_=_#2 & lemma=/(ge)?kommen/ & #3 _=_ #1`	5	5922	5937	5848	237	262
166	`pos="ADJA" & tok=/ge.+en..?/ & #1_=_#2 & lemma=/(ge)?sinken/ & #3 _=_ #1`	0	5943	5941	5848	235	255
167	`pos="ADJA" & tok=/ge.+t..?/ & #1_=_#2 & lemma=/(ge)?führen/ & #3 _=_ #1`	0	5944	5946	5852	240	397
168	`pos="ADJA" & tok=/ge.+en..?/ & #1_=_#2 & lemma=/(ge)?leiden/ & #3 _=_ #1`	0	5954	5937	5862	238	256
169	`pos="ADJA" & tok=/ge.+en..?/ & #1_=_#2 & lemma=/(ge)?wachsen/ & #3 _=_ #1`	13	5962	5970	5875	236	256
170	`pos="ADJA" & tok=/ge.+en..?/ & #1_=_#2 & lemma=/(ge)?brennen/ & #3 _=_ #1`	0	5969	5963	5900	237	262
171	`lemma=/[^äöü]+/ & tok=/.+[äöü].+/ & pos="NN" & #1 _=_ #2 & #2 _=_ #3`	5635	6225	6224	6178	469	6246

Runtime (in ms) of queries in group D.

Query		Results	FROM	WHERE	CTE	MonetDB	PostgreSQL
172	`"kündigten" & "Rias" & #1 .* #2`	1	12	12	12	11	16
173	`"heißt" & "Appell" & #1 .* #2`	3	13	11	12	9.5	19
174	`lemma = "müssen" & lemma = "weg" & #1.*#2`	53	179	866	223	32	28
175	`lemma = "müssen" & lemma = "weg" & #1.#2`	1	229	227	226	33	27
176	`lemma = "müssen" & lemma = "Gefängnis" & #1.*#2`	52	184	879	229	33	29
177	`lemma = "müssen" & lemma = "ins" & #1.*#2`	0	148	865	229	32	18
178	`lemma = "müssen" & lemma = "in" & #1.*#2`	27088	6681	887	255	49	233
179	`pos="VVFIN" & pos="NN" & #1 .1,3 #2`	17064	2087	1924	1825	1584	12766
180	`pos=/de.*/ & pos="VVFIN" & #1 .1,3 #2`	0	2496	2514	2506	82	203
181	`tok = /müssen/ & pos = "$." & #1.*#2`	17175	2890	2897	2958	65	154
182	`/der.*/ & pos="VVFIN" & #1 .2 #2`	1346	3044	3057	3035	63	881
183	`/der.*/ & pos="NN" & #1 . #2`	16383	3087	3086	3084	104	1342
184	`lemma = "müssen" & pos != /VV.*/ & #1.*#2`	1100647	60000	60000	3587	752	7043
185	`pos = /VM.*/ & pos = /VV.*/ & #1.#2`	250	4745	4732	4738	102	226
186	`pos = /VM.*/ & pos = /VV.*/ & #1.*#2`	428175	25964	7805	5063	239	1705
187	`pos=/N.*/ & /[12][09][0-9][0-9]/ & #1 . #2`	1143	5716	5710	5697	246	2903

Runtime (in ms) of queries in group E. Values for PostgreSQL which are larger than 60 seconds are taken from a single run and are not averaged.

Query		Results	FROM	WHERE	CTE	MonetDB	PostgreSQL
188	`pos="KOUS" & "man" & "sich" & #1 . #2 & #2 . #3`	21	67	68	59	53	97
189	`pos = /VM.*/ & pos = /VV.*/ & pos = "$./" & #1 . #2 & #2 .* #3`	0	4762	4742	4785	103	29
190	`pos = /VM.*/ & pos = /VV.*/ & pos = "($.\|$,)" & #1.*#2 & #2.*#3`	0	4744	4755	4853	241	22
191	`pos = /VM.*/ & pos = /VV.*/ & pos = /\$./ & #1.#2 & #2.*#3`	21101	53352	6893	6872	128	504
192	`lemma = "müssen" & pos = /VV.*/ & pos = "$." & #1.*#2 & #2.*#3`	4934027	60000	55427	10099	2045	35898
193	`pos = /VM.*/ & pos = /VV.*/ & pos = /\$.*/ & #1.*#2 & #2.*#3`	53381806	60000	60000	49300	20489	> 1 hour
194	`pos = /VM.*/ & pos = /VV.*/ & pos = /\$./ & #1.*#2 & #2.*#3`	53381806	60000	60000	50605	17136	> 1 hour
195	`pos = /VM.*/ & pos = /VV.*/ & pos = /(\$.|\$,)/ & #1.*#2 & #2.*#3`	53381806	60000	60000	50972	20449	221899
196	`pos = /VM.*/ & pos = /VV.*/ & pos = /\$,/ & #1.*#2 & #2.*#3`	21195563	60000	60000	55153	7835	159253
197	`pos = /VM.*/ & pos = /VV.*/ & pos = "$." & #1.*#2 & #2.*#3`	19880160	60000	60000	59359	7278	90226
198	`lemma = "müssen" & pos != /VV.*/ & pos="$." & #1.*#2 & #2.*#3`	56465383	60000	60000	60000	18303	291707
199	`lemma = "wollen" & pos != /VV.*/ & pos="$." & #1.*#2 & #2.*#3`	32066327	60000	60000	60000	12918	146409
200	`lemma="müssen" & pos!="VV.*" & pos="$." & #1 .* #2 & #2 .* #3`	61399410	60000	60000	60000	19946	258617
201	`pos = /VM.*/ & pos != /VV.*/ & pos="$." & #1.*#2 & #2.*#3`	230726493	60000	60000	60000	113993	968239
202	`pos = /VM.*/ & pos = /VV.*/ & pos = /.*/ & #1.*#2 & #2.*#3`	383753400	60000	60000	60000	210359	3512753

Runtime (in ms) of queries in group F.

Query		Results	FROM	WHERE	CTE	MonetDB	PostgreSQL
203	cat="S" & cat="NP" & cat="NP" & #1 >[func="OA"] #2 & #1 >[func="SB"] #3 & #2 .* #3	412	614	1018	228	198	268
204	cat="S" & "umfaßt" & #1 > #2	29	8211	8303	258	255	16
205	cat="S" & pos="PTKVZ" & pos="VVFIN" & #1 > #3 & #1 _l_ #2	36	577	576	265	188	84
206	cat="S" & "heißt" & "Appell" & #2 .* #3 & #1 > #2	3	189	197	270	264	29
207	cat="S" & **node** & pos="VVFIN" & **node** & #1 >[func="OA"] #2 & #1 > #3 & #1 >[func="SB"] #4 & #2 .* #3 & #3 .* #4	905	967	1040	271	218	600
208	cat="S" & "umfaßt" & "albanischen" & #1 > #2 & #1 > #3	0	297	300	280	272	17
209	cat="S" & "umfaßt" & "albanischen" & #1 >* #2 & #1 >* #3	2	297	6928	280	274	29
210	cat="S" & "umfaßt" & "heißt" & #1 >* #2 & #1 _i_ #3	1	256	252	285	273	188
211	"desto" & "desto" & #1 $* #2 & #1 .* #2	1	296	299	285	270	18
212	cat="S" & "umfaßt" & "heißt" & #1 >* #2 & #1 >* #3	1	323	7069	286	276	32
213	cat="S" & "umfaßt" & "heißt" & #1 _i_ #2 & #1 >* #3	0	251	248	289	277	38
214	cat="S" & #1:**root** & pos="VVFIN" & #1 > #2 & **node** & #1 >[func="SB"] #3 & #2 .* #3	9364			304	214	533
215	pos="NE" & cat="S" & pos="PRELS" & pos="VVFIN" & #2 >[func="HD"] #4 & #1 $ #2 & #3 $ #4	192	1621	60000	353	279	140
216	cat="S" & pos="VVFIN" & cat & cat & #1 >[func="HD"] #2 & #1 _l_ #3 & #3 . #4 & #4 . #2	232	592	598	391	348	955
217	cat="S" & #1:**root** & pos="VVFIN" & #1 >[func="HD"] #2 & cat & #1 > #3 & #1 _l_ #3 & cat & #1 > #4 & #3 . #4 & #4 . #2	8			533	511	619
218	cat="S" & pos="VVFIN" & #1 _i_ #2	45375	12677	26800	988	877	4795
219	cat=/(S|VP)/ & lemma="machen" & #1 >edge #2	810	9171	9229	1567	371	32
220	**tok** & cat="VP" & #1 . #2 & **tok** & #2 _i_ #3 & #1 $ #3	1391	6088	6178	1713	1753	5018
221	/[Jj]e/ & "desto" & #1 $* #2 & morph="Comp" & morph="Comp" & #1 . #3 & #2 . #4	10	11511	9831	3325	284	93
222	**tok**=/be.+/ & pos=/VVFIN/ & #1_i_#2	3250	5195	5194	5190	172	1775
223	cat = "S" & pos = /VM.*/ & pos = /VV.*/ & #1>*#2 & #1>*#3	22104	60000	60000	5202	225	661
224	cat = "S" & pos = /VM.*/ & pos != /VV.*/ & #1>*#2 & #1>*#3	203627	38144	37826	5756	1088	1727

B.2 Regular expressions

The data in the following table was used to generate the chart in section 6.3.4.

Runtime (in ms) of regular expression searches using different optimizations.

	Query	Results	Unopt.	Bounded	BAT	BoundedBAT	Exact	NULLs
1	tok=/gebrannt/	0	2829	36	246	35	4.8	2652
2	tok=/brannte/	4	2808	37	246	38	4.9	2605
3	tok = /müssen/	497	2808	35	247	34	5	2616
4	/kann.*/	886	2912	40	243	35		2695
5	tok = /bekomm../	88	2838	41	246	39		2629
6	/der.*/	27468	2902	140	260	67		2691
7	/de.*/	51189	2865	204	262	76		2649
8	tok=/.+nd..?/	11920	3007		254			2794
9	tok=/.*s.*/	212862	3016		315			2825
10	tok=/.*und.*/	22589	3036		260			2808
11	tok=/.*sich.*/	7686	3048		258			2833
12	/[Kk]ann.*/	905	3135		247			2908
13	lemma=/operiert/	0	2339	140	264	136	46	2417
14	lemma=/besuchen/	63	2331	150	265	145	47	2425
15	lemma=/brannt/	0	2270	151	264	145	47	2365
16	lemma = /gebrannt/	1	2331	148	267	139	47	2425
17	lemma=/gelitten/	1	2334	145	265	142	48	2458
18	lemma=/gewachsen/	19	2367	145	266	142	48	2456
19	lemma = /brennen/	10	2295	148	269	145	49	2394
20	lemma=/gebissen/	0	2331	146	265	138	49	2423
21	lemma=/besuchend/	1	2366	145	267	143	50	2459
22	lemma=/waschen/	8	2324	138	266	133	50	2384
23	lemma=/begonnen/	17	2333	150	266	146	50	2422
24	lemma=/brennt/	0	2278	151	263	147	50	2360
25	lemma=/besetzt/	33	2296	150	269	146	50	2387
26	lemma=/operieren/	16	2367	136	267	136	51	2454
27	lemma=/gesunken/	8	2331	143	262	142	55	2423
28	lemma=/brannt?/	0	2330	149	267	147		2408
29	pos = /VM.*␣.*␣\$./	0	2665	173	238	143		2677
30	pos = /VM.*␣\$./	0	2505	174	236	144		2526
31	pos = /VM.*.*\$./	0	2569	174	237	145		2603
32	pos = /VM.*/	9325	2356	178	279	150		2394
33	pos = /VM.*VV.*.*\$./	0	2771	178	237	141		2802

B.3 Sorted tables

The data in the following tables was used to generate the charts in section 6.4.

Runtime (in ms) of text searches on a sorted `node` table.

	Query	Results	Unsorted	Sorted
1	`tok="gleich"`	108	4.1	3.6
2	`tok=/gebrannt/`	0	4.4	4.4
3	`"jede"`	78	4.6	4.7
4	`tok="begreifen"`	9	4.6	4.5
5	`tok="bewerten"`	11	4.6	4.8
6	`"jedes"`	59	4.7	4.5
7	`"man"`	974	4.7	4.6
8	`"Rias"`	2	4.7	4.9
9	`tok = /müssen/`	497	4.7	4.7
10	`tok="bedeuten"`	31	4.7	4.5
11	`tok="befassen"`	14	4.7	4
12	`tok="berichten"`	25	4.7	4.3
13	`"was"`	502	4.8	4.5
14	`tok="befürchten"`	38	4.8	4.3
15	`tok="beginnen"`	34	4.8	4.5
16	`tok=/brannte/`	4	4.8	4.5
17	`"alles"`	210	4.9	4.3
18	`"Appell"`	17	5.2	5.3
19	`"6"`	24	5.7	5.4
20	`"das"`	6082	5.8	5
21	`tok=".+nd.?.?"`	0	6.2	5.2
22	`"der"`	26779	9.7	5.1
23	`"die"`	24467	10	5
24	`/kann.*/`	886	35	7.2
25	`tok = /bekomm../`	88	38	6.6
26	`/der.*/`	27468	56	20
27	`/de.*/`	51189	66	29
28	`/[Kk]ann.*/`	905	244	220
29	`tok=/.+nd..?/`	11920	249	208
30	`tok=/.*und.*/`	22589	251	212
31	`tok=/.*sich.*/`	7686	254	216
32	`tok=/.*s.*/`	212862	274	231

Runtime (in ms) of annotation searches on a sorted `node_annotation` table.

Query		Results	Unsorted	Sorted on name	Sorted on value
1	`word="laufend..?"`	0	15	15	15
2	`word=".+nd.?.?"`	0	15	17	15
3	`word="laufend?.?"`	0	16	17	14
4	`word="schlafend..?"`	0	16	18	16
5	`lemma`	888578	27	38	37
6	`cat="T"`	0	33	32	34
7	`cat="S"`	72346	34	39	32
8	`cat="TP"`	0	37	32	32
9	`pos="VVIFIN"`	0	48	47	48
10	`lemma="--"`	121701	50	49	43
11	`pos="CARD"`	15939	50	48	47
12	`lemma=/besetzt/`	33	50	48	47
13	`pos="VVIMP"`	162	50	46	46
14	`pos="VVPP"`	17770	50	44	47
15	`pos= "ADJA"`	54534	51	47	46
16	`lemma=/operieren/`	16	51	50	46
17	`pos="ADJ."`	0	51	46	47
18	`lemma=/operiert/`	0	51	48	44
19	`lemma=/besuchen/`	63	52	48	45
20	`lemma=/begonnen/`	17	52	48	46
21	`lemma=/gelitten/`	1	52	48	46
22	`pos="VVFIN"`	35628	52	48	45
23	`lemma = /brennen/`	10	52	47	48
24	`lemma = /gebrannt/`	1	52	48	49
25	`lemma=/brennt/`	0	52	49	47
26	`lemma=/gesunken/`	8	52	49	46
27	`lemma=/waschen/`	8	52	49	47
28	`lemma=/gebissen/`	0	53	49	47
29	`lemma="fragen"`	98	53	47	49
30	`lemma=/brannt/`	0	54	47	48
31	`lemma = "müssen"`	1880	54	47	45
32	`pos="ADJ"`	0	54	48	46
33	`lemma=/besuchend/`	1	54	49	49
34	`lemma=/gewachsen/`	19	56	48	47
35	`pos = /VM.*VV.*.*\$./`	0	143	57	125
36	`pos = /VM.*.*\$./`	0	145	55	123
37	`pos = /VM.*␣\$./`	0	146	56	122
38	`pos = /VM.*␣.*␣\$./`	0	148	55	124
39	`pos = /VM.*/`	9325	148	58	126
40	`lemma=/brannt?/`	0	151	33	138

References

[A$^+$03] Stefanie Albert et al. TIGER Annotationsschema. Technical report, Universität des Saarlandes, Universität Stuttgart, and Universität Potsdam, 2003.

[AAB$^+$08] Rakesh Agrawal, Anastasia Ailamaki, Philip A. Bernstein, Eric A. Brewer, Michael J. Carey, Surajit Chaudhuri, AnHai Doan, Daniela Florescu, Michael J. Franklin, Hector Garcia-Molina, Johannes Gehrke, Le Gruenwald, Laura M. Haas, Alon Y. Halevy, Joseph M. Hellerstein, Yannis E. Ioannidis, Hank F. Korth, Donald Kossmann, Samuel Madden, Roger Magoulas, Beng Chin Ooi, Tim O'Reilly, Raghu Ramakrishnan, Sunita Sarawagi, Michael Stonebraker, Alexander S. Szalay, and Gerhard Weikum. The Claremont Report on Database Research. *SIGMOD Record*, 37(3):9–19, 2008.

[ABdVB06] Wouter Alink, Raoul Bhoedjang, Arjen de Vries, and Peter Boncz. Efficient XQuery Support For Stand-Off Annotation. In *Proceedings of International Workshop on XQuery Implementation, Experience and Perspectives*, 2006.

[ABH09] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. Column-oriented Database Systems. In *Proceedings of the VLDB Endowment*, August 2009.

[ADHW99] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *Proceedings of the International Conference on Very Large Data Bases*, 1999.

[BCD$^+$05] Steven Bird, Yi Chen, Susan Davidson, Haejoong Lee, and Yifeng Zheng. Extending XPath to Support Linguistic Queries. In *Proceedings of the Workshop on Programming Language Technologies for XML*, 2005.

[BCD$^+$06] Steven Bird, Yi Chen, Susan B. Davidson, Haejoong Lee, and Yifeng Zheng. Designing and Evaluating an XPath Dialect for Linguistic Queries. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2006.

[BCF$^+$10] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, Jérôme Siméon, et al. XQuery 1.0: An XML Query Language (Second Edition) [online]. 2010. Available from: http://www.w3.org/TR/xquery [cited 2012/08/31].

[BCR98] Douglas Biber, Susan Conrad, and Randi Reppen. *Corpus Linguistics: Investigating Language Structure and Use.* Cambridge University Press, 1998.

[BDE$^+$04] Sabine Brants, Stefanie Dipper, Peter Eisenberg, Silvia Hansen-Schirra, Esther König, Wolfgang Lezius, Christian Rohrer, George Smith, and Hans Uszkoreit. TIGER: Linguistic Interpretation of a German Corpus. *Research on Language & Computation*, 2(4):597–620, 2004.

[BGvK$^+$06] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2006.

[BJL$^+$99] Douglas Biber, Stig Johansson, Geoffrey Leech, Susan Conrad, Edward Finegan, and Randolph Quirk. *Longman Grammar of Spoken and Written English.* MIT Press, 1999.

[BK99] Peter A. Boncz and Martin L. Kersten. MIL primitives for querying a fragmented world. *The VLDB Journal*, 1999.

[BK02] Gosse Bouma and Geert Kloosterman. Querying Dependency Treebanks in XML. In *Proceedings of the International Conference on Language Resources and Evaluation*, 2002.

[BK07] Gosse Bouma and Geert Kloosterman. Mining Syntactically Annotated Corpora with XQuery. In *Proceedings of the Linguistic Annotation Workshop*, 2007.

[BL07] Steven Bird and Haejoong Lee. Graphical Query for Linguistic Treebanks. In *Proceedings of the Conference of the Pacific Association for Computational Linguistics*, 2007.

[BMK99] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proceedings of the International Conference on Very Large Data Bases*, 1999.

[BPSM$^+$00] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, et al. Extensible Markup Language (XML) 1.0 [online]. 2000. Available from: http://www.w3.org/TR/REC-xml [cited 2012/08/31].

[BR08] Ute Bohnacker and Christina Rosén. How to start a V2 declarative clause: Transfer of syntax vs. information structure in L2 German. *Nordlyd*, 34(3):29–56, 2008.

[BRK98] Peter A. Boncz, Tim Rühl, and Fred Kwakkel. The Drill Down Benchmark. In *Proceedings of the International Conference on Very Large Data Bases*, 1998.

[BZ10] Margit Breckle and Heike Zinsmeister. A corpus-based contrastive analysis of local coherence in L1 and L2 German. In *Proceedings of the International Conference by the Croatian Applied Linguistics Society*, 2010.

[C$^+$99] James Clark et al. XSL Transformations (XSLT) Version 1.0 [online]. 1999. Available from: http://www.w3.org/TR/xslt [cited 2012/08/31].

[Cas02] Steve Cassidy. XQuery as an Annotation Query Language: a Use Case Analysis. In *Proceedings of the Language Resources and Evaluation Conference*, 2002.

[CD$^+$99] James Clark, Steve DeRose, et al. XML Path Language (XPath) Version 1.0 [online]. 1999. Available from: http://www.w3.org/TR/xpath [cited 2012/08/31].

[CEHK05] Jean Carletta, Stefan Evert, Ulrich Heid, and Jonathan Kilgour. The NITE XML Toolkit: Data Model and Query Language. *Language Resources and Evaluation*, 39(4):313–334, 2005.

[CHSSZE05] Berthold Crysmann, Silvia Hansen-Schirra, George Smith, and Dorothea Ziegler-Eisele. TIGER Morphologie-Annotationsschema. Technical report, Universität des Saarlandes, Universität Stuttgart, and Universität Potsdam, 2005.

[CK85] George P. Copeland and Setrag N. Khoshafian. A Decomposition Storage Model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1985.

[CKPS95] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. In *Proceedings of the International Conference on Data Engineering*, 1995.

[Cro93] Steve Crowdy. Spoken Corpus Design. *Literary and Linguistic Computing*, 8(4):259–265, 1993.

[CRS11] Christian Chiarcos, Julia Ritz, and Manfred Stede. Querying and visualizing coreference annotation in multi-layer corpora. In *Proceedings of the Discourse Anaphora and Anaphor Resolution Colloquium*, 2011.

[DGSW04] Stefanie Dipper, Michael Götze, Manfred Stede, and Tillmann Wegst. ANNIS: A Linguistic Database for Exploring Information Structure. Technical report, Universität Potsdam, 2004.

[Dra37] Erich Drach. *Grundgedanken der deutschen Satzlehre*. M. Diesterweg, 1937.

[ET07] Richard Eckart and Elke Teich. An XML-based data model for flexible representation and query of linguistically interpreted corpora. In *Proceedings of the Biannual Conference of the Society for Computational Linguistics and Language Technology*, 2007.

[Gar95] Stephen R. Garner. WEKA: The Waikato Environment for Knowledge Analysis. In *Proceedings of New Zealand Computer Science Research Students Conference*, 1995.

[GHMP04] Dennis F. Galletta, Raymond Henry, Scott McCoy, and Peter Polak. Web Site Delays: How Tolerant are Users? *Journal of the Association for Information Systems*, 5(1):1, 2004.

[GKT04] Torsten Grust, Maurice Van Keulen, and Jens Teubner. Accelerating XPath Evaluation in Any RDBMS. *ACM Transactions on Database Systems*, 29(1):91–131, 2004.

[GMS92] Hector Garcia-Molina and Kenneth Salem. Main Memory Database Systems: An Overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.

[H$^+$97] Philip Hazel et al. PCRE – Perl Compatible Regular Expressions [online]. 1997. Available from: http://www.pcre.org [cited 2012/08/15].

[HBK$^+$00] Edward W. Hinrichs, Julia Bartels, Yasuhiro Kawata, Valia Kordoni, and Heike Telljohann. The VERBMOBIL Treebanks. In *Proceedings of the Conference on Natural Language Processing (KONVENS)*, 2000.

[HKN$^+$04] Erhard Hinrichs, Sandra Kübler, Karin Naumann, Heike Telljohann, Julia Trushkina, et al. Recent Developments in Linguistic Annotations of the TüBa-D/Z Treebank. In *Proceedings of the Workshop on Treebanks and Linguistic Theories*, 2004.

[HRŠŠ10] Jirka Hana, Alexandr Rosen, Svatava Škodová, and Barbora Štindlová. Error-tagged Learner Corpus of Czech. In *Proceedings of the Linguistic Annotation Workshop*, 2010.

[Hüt08] Karsten Hütter. Entwicklung einer Benutzerschnittstelle für die Suche in linguistischen mehrebenen Korpora unter Betrachtung softwareergonomischer Gesichtspunkte. Diplomarbeit. Humboldt-Universität zu Berlin, 2008.

[IEE04] IEEE. *IEEE Standard for Information Technology – Portable Operating System Interface (POSIX). Shell and Utilities. IEEE standard 1003.2-2004*. IEEE Computer Society, 2004.

[INGK07] Milena Ivanova, Niels Nes, Romulo Goncalves, and Martin L. Kersten. MonetDB/SQL Meets SkyServer: the Challenges of a Scientific Database. In *Proceedings of the International Conference on Scientific and Statistical Database Management*, 2007.

[ISO03] ISO/IEC. *Information technology – Database languages – SQL, ISO/IEC standard 9075-*:2003*. International Organisation for Standardization / International Electrotechnical Commission, 2003.

[Kep03] Stephan Kepser. Finite Structure Query: A Tool for Querying Syntactically Annotated Corpora. In *Proceedings of the Conference of the European Chapter of the Association for Computational Linguistics*, 2003.

[Kep04] Stephan Kepser. A Simple Proof for the Turing-Completeness of XSLT and XQuery. In *Proceedings of Extreme Markup Languages*, 2004.

[KM01] Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, Aarhus University, 2001.

[KRZZ11] Thomas Krause, Julia Ritz, Amir Zeldes, and Florian Zipser. Topological Fields, Constituents and Coreference: A New Multi-layer Architecture for TüBa-D/Z. In *Jahrestagung der Gesellschaft für Sprachtechnologie und Computerlinguistik (GSCL)*, 2011.

[LB04] Catherine Lai and Steven Bird. Querying and Updating Treebanks: A Critical Survey and Requirements Analysis. In *Proceedings of the Australasian Language Technology Workshop*, 2004.

[LB10] Catherine Lai and Steven Bird. Querying Linguistic Trees. *Journal of Logic, Language and Information*, 19:53–73, 2010.

[LDH$^+$08] Anke Lüdeling, Seanna Doolittle, Hagen Hirschmann, Karin Schmidt, and Maik Walter. Das Lernerkorpus Falko. *Deutsch als Fremdsprache*, 45:67–73, 2008.

[Lez02] Wolfgang Lezius. *Ein Suchwerkzeug für syntaktisch annotierte Textkorpora*. PhD thesis, Universität Stuttgart, 2002.

[Lüd11] Anke Lüdeling. Corpora in Linguistics: Sampling and Annotation. In *Going Digital: Evolutionary and Revolutionary Aspects of Digitization (Nobel Symposium 147)*, 2011.

[LZ08] Anke Lüdeling and Amir Zeldes. Three Views on Corpora: Corpus Linguistics, Literary Computing, and Computational Linguistics. *Jahrbuch für Computerphilologie*, 2008.

[Mar04] Maarten Marx. XPath with Conditional Axis Relations. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2004.

[Mei03] Wolfgang Meier. eXist: An Open Source Native XML Database. In *Web, Web-Services, and Database Systems*, volume 2593 of *Lecture Notes in Computer Science*. Springer, 2003.

[MIM$^+$01] David McKelvie, Amy Isard, Andreas Mengel, Morten Baun Møller, Michael Grosse, and Marion Klein. The MATE workbench – An annotation tool for XML coded speech corpora. *Speech Communication*, 33(1–2):97–112, 2001.

[MK09] Hendrik Maryns and Stephan Kepser. MonaSearch – Querying Linguistic Treebanks with Monadic Second-Order Logic. In *Proceedings of the International Workshop on Treebanks and Linguistic Theories*, 2009.

[MKB09] Stefan Manegold, Martin L. Kersten, and Peter A. Boncz. Database Architecture Evolution: Mammals Flourished Long Before Dinosaurs Became Extinct. In *Proceedings of the International Conference on Very Large Data Bases*, 2009.

[MKC06] Neil Mayo, Jonathan Kilgour, and Jean Carletta. Towards an Alternative Implementation of NXT's Query Language via XQuery. In *Proceedings of the Workshop on NLP and XML*, 2006.

[MLV08] Torsten Marek, Joakim Lundborg, and Martin Volk. Extending the TIGER Query Language with Universal Quantification. In *Proceedings of the Conference on Natural Language Processing (KONVENS)*, 2008.

[MMS93] Mitchel P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a Large Annotated Corpus of English: the Penn Treebank. *Computational Linguistics - Special issue on using large corpora: II*, 19(2):313–330, 1993.

[MMWK10] Ashok Malhotra, Jim Melton, Norman Walsh, and Michael Kay. XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition) [online]. 2010. Available from: http://www.w3.org/TR/xpath-functions [cited 2012/08/31].

[MRM04] Adam Meyers, Ruth Reeves, and Catherine Macleod. NP-external arguments a study of argument sharing in English. In *Proceedings of the Workshop on Multiword Expressions: Integrating Processing*, 2004.

[PDL$^+$08] Rashmi Prasad, Nikhil Dinesh, Alan Lee, Eleni Miltsakaki, Livio Robaldo Aravind Joshi, and Bonnie Webber. The Penn Discourse TreeBank 2.0. In *Proceedings of the International Conference on Language Resources and Evaluation (LREC)*, 2008.

[PGK05] Martha Palmer, Daniel Gildea, and Paul Kingsbury. The Proposition Bank: An Annotated Corpus of Semantic Roles. *Computational Linguistics*, 31(1):71–106, 2005.

[PIS$^+$05] James Pustejovsky, Robert Ingria, Roser Saurí, José Castaño, Jessica Littman, Rob Gaizauskas, Andrea Setze, Graham Katz, and Inderjeet Mani. The Specification Language TimeML. *The Language of Time: A Reader*, 2005.

[PMPP05] James Pustejovsky, Adam Meyers, Martha Palmer, and Massimo Poesio. Merging PropBank, NomBank, TimeBank, Penn Discourse Treebank and Coreference. In *Proceedings of the Workshop on Frontiers in Corpus Annotations II: Pie in the Sky*, 2005.

[Pos96] PostgreSQL Global Development Group. PostgreSQL. http://www.postgresql.org/, 1996.

[Pos12] PostgreSQL Global Development Group. PostgreSQL Manual: 18.4. Resource Consumption [online]. 2012. Available from: http://www.postgresql.org/docs/9.1/static/runtime-config-resource.html [cited 2012/08/23].

[PV98] Massimo Poesio and Renata Vieira. A corpus-based investigation of definite description use. *Computational Linguistics*, 24(2):183–216, 1998.

[R11] R Development Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2011. ISBN 3-900051-07-0.

[RE05] Philip Resnik and Aaron Elkiss. The Linguist's Search Engine: An Overview. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 2005.

[RECD08] Georg Rehm, Richard Eckart, Christian Chiarcos, and Johannes Dellert. Ontology-Based XQuery'ing of XML-Encoded Language Resources on Multiple Annotation Layers. In *Proceedings of Language Resources and Evaluation Conference (LREC)*, 2008.

[RLH97] Paul Rayson, Geoffrey Leech, and Mary Hodges. Social differentiation in the use of English vocabulary: some analyses of the Conversational Component of the British National Corpus. *International Journal of Corpus Linguistics*, 1997.

[Roh05] Douglas L. T. Rohde. *TGrep2 User Manual*, 2005.

[Ros11] Viktor Rosenfeld. An Implementation Of The Annis 2 Query Language. Studienarbeit, Humboldt-Universität zu Berlin, Available at: http://www.informatik.hu-berlin.de/forschung/gebiete/wbi/teaching/studienDiplomArbeiten/finished/2010/rosenfeld_studienarbeit.pdf, 2011.

[RSW+09] Georg Rehm, Oliver Schonefeld, Andreas Witt, Erhard W. Hinrichs, and Marga Reis. Sustainability of annotated resources in linguistics: A web-platform for exploring, querying, and distributing linguistic corpora and other resources. *Literary and Linguistic Computing*, 24(2):193–210, 2009.

[SBÇ+07] Michael Stonebraker, Chuck Bear, Uğur Çetintemel, Mitch Cherniack, Tingjian Ge, Nabil Hachem, Stavros Harizopoulos, John Lifter, Jennie Rogers, and Stan Zdonik. One Size Fits All? – Part 2: Benchmarking Results. In *Proceedings of the Conference on Innovative Data Systems Research*, 2007.

[Sch03] Ulrich Schäfer. WHAT: An XSLT-based Infrastructure for the Integration of Natural Language Processing Components. In *Proceedings of the Workshop on Software Engineering and Architecture of Language Technology Systems (HLT-NAACL)*, 2003.

[SJ08] Said Sahel and Julia Jonischkait. Syntaktische Funktionen im Vorfeld. Eine empirische Studie. *Muttersprache*, 118(4):281–294, 2008.

[SK02] Ilona Steiner and Laura Kallmeyer. VIQTORYA — A Visual Query Tool for Syntactically Annotated Corpora. In *Proceedings of the International Conference on Language Resources and Evaluation (LREC)*, 2002.

[Spe08] Augustin Speyer. German Vorfeld-filling as constraint interaction. In Anton Benz and Peter Kühnlein, editors, *Constraints in Discourse*. John Benjamins Publishing Company, 2008.

[Spe10] Augustin Speyer. Filling the German vorfeld in written and spoken discourse. In Sanna-Kaisa Tanskanen, Marja-Liisa Helasvuo, Marjut Johansson, and Mia Raitaniemi, editors, *Discourses in Interaction*. John Benjamins Publishing Company, 2010.

[SR03] Jasmin Saric and Uwe Reyle. TIGERSearch attacks Proteins. In *Proceedings of the European Conference on Computational Biology (ECCB)*, 2003.

[Ste04] Manfred Stede. The Potsdam Commentary Corpus. In *Proceedings of the ACL Workshop on Discourse Annotation*, 2004.

[STST99] Anne Schiller, Simone Teufel, Christine Stöckert, and Christine Thielen. Guidelines für das Tagging deutscher Textkorpora mit STTS. Technical report, Universität Stuttgart and Universität Tübingen, 1999.

[Tay03] Claire Louise Taylor. XSLT as a Linguistic Query Language. Master's thesis, The University of Melbourne, 2003.

[Tay08] Charlotte Taylor. What is corpus linguistics? What the data says. *International Computer Archive of Modern and Medieval English*, 32:179–200, 2008.

[TL05] Silke Trißl and Ulf Leser. Querying Ontologies in Relational Database Systems. In *Proceedings of the Workshop on Data Integration in the Life Sciences*, 2005.

[TM97] Henry S. Thompson and David McKelvie. Hyperlink semantics for standoff markup of read-only documents. In *Proceedings of SGML Europe*, 1997.

[VEKC03] Holger Voormann, Stefan Evert, Jonathan Kilgour, and Jean Carletta. *NXT Search User's Manual (Draft)*, 2003.

[Vit04] Thorsten Vitt. Speicherung linguistischer Korpora in Datenbanken. Studienarbeit, Humboldt-Universität zu Berlin. Available at: http://www2.informatik.hu-berlin.de/Forschung_Lehre/wbi/research/stud_arbeiten/finished/2004/vitt_041114.pdf, 2004.

[Vit05] Thorsten Vitt. DDDquery: Anfragen an komplexe Korpora. Diplomarbeit, Humboldt-Universität zu Berlin, 2005.

[VL02] Holger Voormann and Wolfgang Lezius. TIGERin – Grafische Eingabe von Benutzeranfragen für ein Baumbank-Anfragewerkzeug. In *Proceedings of the Conference on Natural Language Processing (KONVENS)*, 2002.

[WHM+11] Ralph Weischedel, Eduard Hovy, Mitchell Marcus, Martha Palmer, Robert Belvin, Sameer Pradhan, Lance Ramshaw, and Nianwen Xue. OntoNotes: A Large Training Corpus for Enhanced Processing. In *Handbook of Natural Language Processing and Machine Translation*. Springer, 2011.

[WN00] Sean Wallis and Gerald Nelson. Exploiting Fuzzy Tree Fragment Queries in the Investigation of Parsed Corpora. *Literary and Linguistic Computing*, 15(3):339–362, 2000.

[ZHS+97] Gisela Zifonun, Ludger Hoffmann, Bruno Strecker, et al. *Grammatik der deutschen Sprache*. Walter de Gruyter, 1997.

[ZR10] Florian Zipser and Laurent Romary. A model oriented approach to the mapping of annotation formats using standards. In *Workshop on Language Resource and Language Technology Standards*, 2010.

[ZRLC09] Amir Zeldes, Julia Ritz, Anke Lüdeling, and Christian Chiarcos. ANNIS: A Search Tool for Multi-Layer Annotated Corpora. In *Proceedings of Corpus Linguistics*, 2009.