An Implementation Of The Annis 2 Query Language

Viktor Rosenfeld[*]
Supervisor: Ulf Leser

April 23, 2010

We describe the Annis 2 Query Language and show how its features including operations on distinct graphs over the same nodes can be implemented using a relational database as a back-end. We provide a reference implementation on top of PostgreSQL and measure its performance on consumer hardware.

---

[*] rosenfel@informatik.hu-berlin.de

Contents

List of Figures

List of Tables

Listings

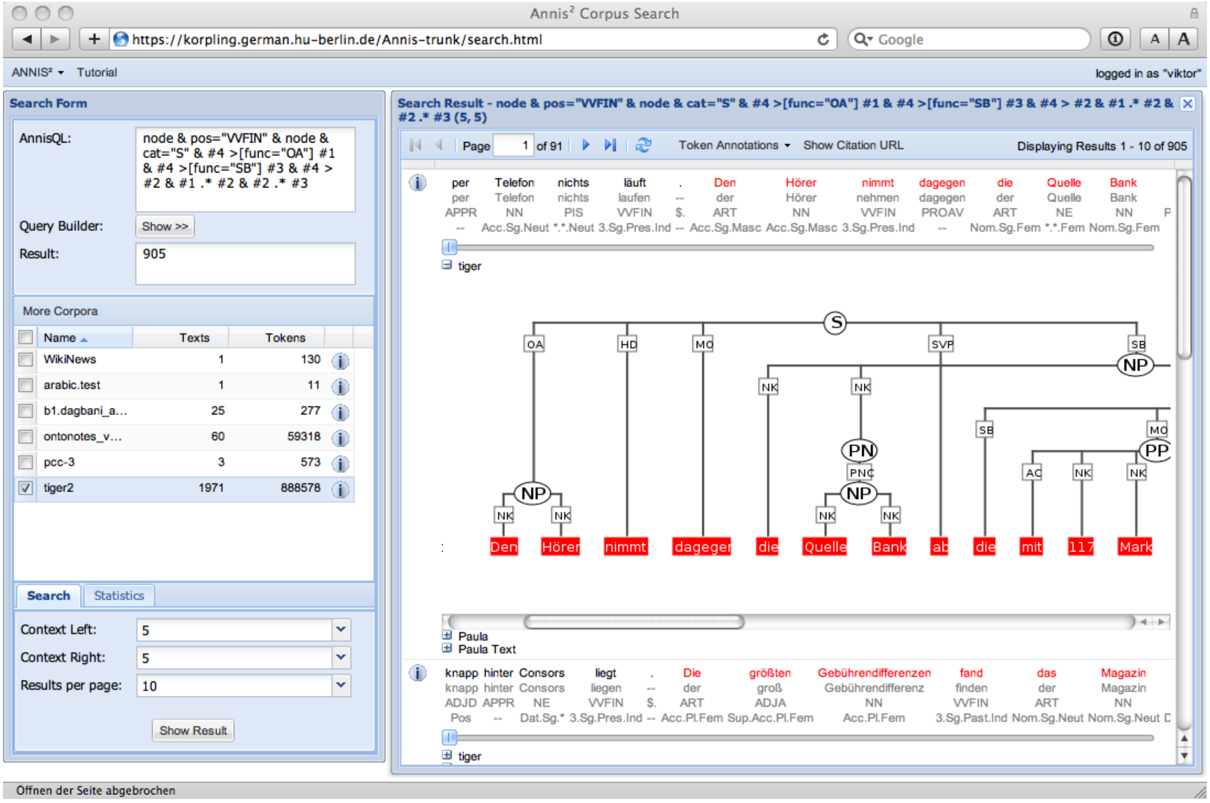
1 Introduction

Annis 2 is a search engine and visualization tool for linguistic text corpora containing conflicting, multi-modal annotations over the same texts [25]. Annotations can be key-value pairs attached to text spans including support for multimedia elements, syntax graphs over a set of tokens in a text or arbitrary links between spans. This work primarily discusses the Annis 2 back-end – the part of the system that translates Annis queries into SQL. Its main contribution is support for distinct graph types over the same nodes, i.e. the ability to construct graphs containing different types of edges over a set of tokens and query for them either separately or collectively. We use this feature to implement both dominance and arbitrary pointing relationships between text spans.

1.1 Historical overview of Annis

A predecessor of the system – now called Annis 1 – was developed within the Sonderforschungsbereich 632 at the University of Potsdam [25]. It became apparent that Annis 1 was not particularly suited for large corpora because of its in-memory architecture and a desire to integrate the system with a database emerged. At the same time, the Humboldt-University of Berlin was developing a SQL compiler for DDD*query*, the query language used by the DeutschDiachronDigital project [29, 2]. In an effort to reuse code, a simple mapping from the Annis 1 query language to DDD*query* was devised, so Annis 1 could support a database back-end as fast as possible.

The time spent during the database port was used to simplify the Annis query language and extend it with new features. The project was also joined by Karsten Hütter who developed a web frontend for a linguistic search engine including an advanced AJAX query builder as part of his diploma thesis [18]. Today his work, of which a screen shot is shown in Figure 1, is the most visible part of the Annis 2 system as the user interacts with it directly.



https://korpling.german.hu-berlin.de/Annis-trunk/search.html

logged in as "viktor"

AnnisQL: node & pos="VVFIN" & node & cat="S" & #4 >[func="OA"] #1 & #4 >[func="SB"] #3 & #4 > #2 & #1 .* #2 & #2 .* #3

Result: 905

☐	Name	Texts	Tokens
☐	WikiNews	1	130
☐	arabic.test	1	11
☐	b1.dagbani_a...	25	277
☐	ontonotes_v...	60	59318
☐	pcc-3	3	573
☑	tiger2	1971	888578

Context Left: 5

Context Right: 5

Results per page: 10

Search Result - node & pos="VVFIN" & node & cat="S" & #4 >[func="OA"] #1 & #4 >[func="SB"] #3 & #4 > #2 & #1 .* #2 & #2 .* #3 (5, 5)

Page 1 of 91

Displaying Results 1 - 10 of 905

per	Telefon	nichts	läuft	.	Den	Hörer	nimmt	dagegen	die	Quelle	Bank
per	Telefon	nichts	laufen	--	der	Hörer	nehmen	dagegen	der	Quelle	Bank
APPR	NN	PIS	VVFIN	$.	ART	NN	VVFIN	PROAV	ART	NE	NN
--	Acc.Sg.Neut	*.*.Neut	3.Sg.Pres.Ind	--	Acc.Sg.Masc	Acc.Sg.Masc	3.Sg.Pres.Ind	--	Nom.Sg.Fem	*.*.Fem	Nom.Sg.Fem

tiger

knapp	hinter	Consors	liegt	.	Die	größten	Gebührendifferenzen	fand	das	Magazin
knapp	hinter	Consors	liegen	--	der	groß	Gebührendifferenz	finden	der	Magazin
ADJD	APPR	NE	VVFIN	$.	ART	ADJA	NN	VVFIN	ART	NN
Pos	--	Dat.Sg.*	3.Sg.Pres.Ind	--	Acc.Pl.Fem	Sup.Acc.Pl.Fem	Acc.Pl.Fem	3.Sg.Past.Ind	Nom.Sg.Neut	Nom.Sg.Neut

tiger

Figure 1: Screenshot of the Annis 2 web application with a result for the query example in section 3.

1.2 Goals and structure of this work

The goals of this work are to formally define the concepts used within Annis 2, to develop an implementation of the Annis 2 Query Language (AQL2) on top of a relational database host (RDBMS) that can be used interactively with large corpora, to study optimization techniques, and to provide detailed performance measurements of the entire system.

We first provide a formal definition of the corpus model used by Annis including a mapping to a SQL schema in section 2. Then, in section 3 we discuss the features of the Annis 2 Query Language including query functions. In section 4 we provide a reference implementation of AQL2 on top of the open-source RDBMS PostgreSQL. This section includes a detailed description on how graphs with multiple edge types can efficiently be supported on SQL hosts. We briefly discuss related work on evaluating XPath queries on relational databases and contrast Annis with TIGERSearch in section 5. The system is evaluated in section 6 for its performance on current consumer hardware on a moderately large corpus. Finally, in section 7 we summarize our findings and discuss on-going and future work on Annis.

In the appendix we briefly discuss the Internal DDD*query* implementation and provide an Annis 2 Query Language Grammar, the SQL Schema of the Corpus Data Model and a description of the Experimental Setup including information on the TIGER corpus.

2 Corpus Data Model

2.1 Overview

The corpus data model defines a normalized representation of the information contained in an annotated corpus. It is used as an intermediary format to import the information generated by various annotation tools into the Annis service. During an import it is augmented with pre-computed, index-like information to implement different operations on the corpus data.

Informally a corpus consists of one or more texts (*primary data*), such as a newspaper article, and annotations on these texts (*secondary data*). Individual *text spans* are modeled as nodes which are arranged in an ordered directed acyclic graph (ODAG). For each text an ordered subset of nodes define the *tokens* of the text. Edges between nodes can carry arbitrary semantic meaning. Currently we distinguish between edges that encode *coverage*, *dominance* and *pointing relations* between text spans. Nodes, edges and corpora can be annotated with *key-value pairs*.

A corpus can also contain child corpora (called *documents*) which are arranged in a hierarchy.

2.2 Key concepts

Definition 1 (Primary data) *Any raw text can be used as* primary data *for a corpus. A primary data text has a unique identifier* $\mathsf{id}_{\mathsf{text}}$ *and an informational* name*.*

Definition 2 (Text span) *The triplet* $(\mathsf{id}_{\mathsf{text}}, \mathsf{left}, \mathsf{right})$ *defines a* text span*. It is the substring from* left *to* right *(inclusive) of the primary data text identified by* $\mathsf{id}_{\mathsf{text}}$*. Both* left *and* right *refer to character positions of the primary data text, starting with 0.*

Definition 3 (Token) *Let t be a primary text and S a set of spans from t. A subset $T \subseteq S$ is called the* tokens of t *if the following conditions hold:*

1. *T is well-ordered under a relation $\leq_{\text{pos}}$,*
2. $\forall i, j \in T : i <_{\text{pos}} j \Rightarrow i_{\text{right}} < j_{\text{left}}$*,*
3. $\forall i \in T : \neg\exists j \in S : (i_{\text{left}} < j_{\text{left}} < i_{\text{right}}) \vee (i_{\text{left}} < j_{\text{right}} < i_{\text{right}})$*, and*
4. $\forall i, j \in T : (i <_{\text{pos}} j \wedge \neg\exists k \in T : i <_{\text{pos}} k <_{\text{pos}} j) \Rightarrow \neg\exists l \in S : i_{\text{right}} \leq l_{\text{left}} \wedge l_{\text{right}} \leq j_{\text{left}}$

Informally, the token order relation $\leq_{\text{pos}}$ does not contradict the order implied by the position of the spans as substrings of the text t (2); if i is a token then there exists no span with its left or right border within i (3); and if i and j are two consecutive tokens then there exists no span between them (4).

The reasoning behind these admittedly complex requirements is that although Annis supports conflicting annotations by different tools over the same text, all annotation features must refer to a shared token layer.

Definition 4 (Annotation graph) *Let T be a set of primary data texts. An* annotation graph over T *is an ordered directed acyclic graph of text spans taken from T.*

A node is defined by the tuple $(\mathsf{span}, \mathsf{name}, \mathsf{annotations}, \mathsf{continuous})$ *where*

- span *is a text span,*
- name *is an informational name, optionally qualified with a namespace,*
- annotations *is a set of key-value pair annotations and*
- continuous *specifies whether the text span is gap-free or not.*

An edge is defined by the tuple $(\mathsf{source}, \mathsf{destination}, \mathsf{type}, \mathsf{name}, \mathsf{annotations})$ *where*

- source *and* destination *are the edge's nodes,*

- type *is a label that encodes the type of the edge,*
- name *is a label that partitions edges of a given type, optionally qualified with a namespace and*
- annotations *is a set of key-value pair annotations.*

The distinction between edge type and name is an artifact of the query language which is discussed in the next section. The edge type is determined by a linguistic constraint between two text spans. The constraint can be qualified with an edge name to only select some of the edges it normally operates on.

Currently three different types of edges are supported:

- *Coverage* edges from a parent span to two or more child spans group the child spans, allowing the construction of text spans with gaps in them.
- *Dominance* edges encode the syntax structure of text spans. Note that dominance implies coverage but not vice versa.
- *Pointing relation* edges encode semantic relations between text spans.

Note that each span of a primary text can be referred to by more than one node in the annotation graph. It is also worth pointing out that tokens are not necessarily leafs in the annotation graph. A trivial example of a token represented by a non-terminal node in the graph is a token with an outgoing pointing relation edge.

Definition 5 (Document, Corpus) *A* document *is defined by the tuple* (name, texts, graph, annotations) *where*

- name *is a informational name that has to be unique for root documents,*
- texts *is a set of primary data texts,*
- graph *is an annotation graph over* texts*,*
- annotations *is a set of key-value pair annotations (used as meta data).*

Documents are arranged in a hierarchy with multiple roots. A root document is called a corpus*.*

2.3 SQL schema

In this section we will develop a SQL schema for (a variant of) the corpus data model. The schema shown in Figure 2 represents the output format of the Annis converter.[1] It is a very close adaptation of the corpus data model with one particularity: for token spans the covered text is stored in the node representing the span. The schema is described in detail below. In order to keep things simple we will omit a few details, such as UNIQUE constraints. The complete schema, including the modifications made during import, is listed in appendix C.

Documents and corpora are stored in a table corpus using the combined pre- and post-order scheme (see below) to encode the document hierarchy. Meta data is stored in the table corpus_annotation.

corpus:

id	primary key
name	name of the corpus
pre	pre-order value
post	post-order value

---

[1] The Annis converter transforms data files of different linguistic tools to the relational format expected by Annis [32, 33].

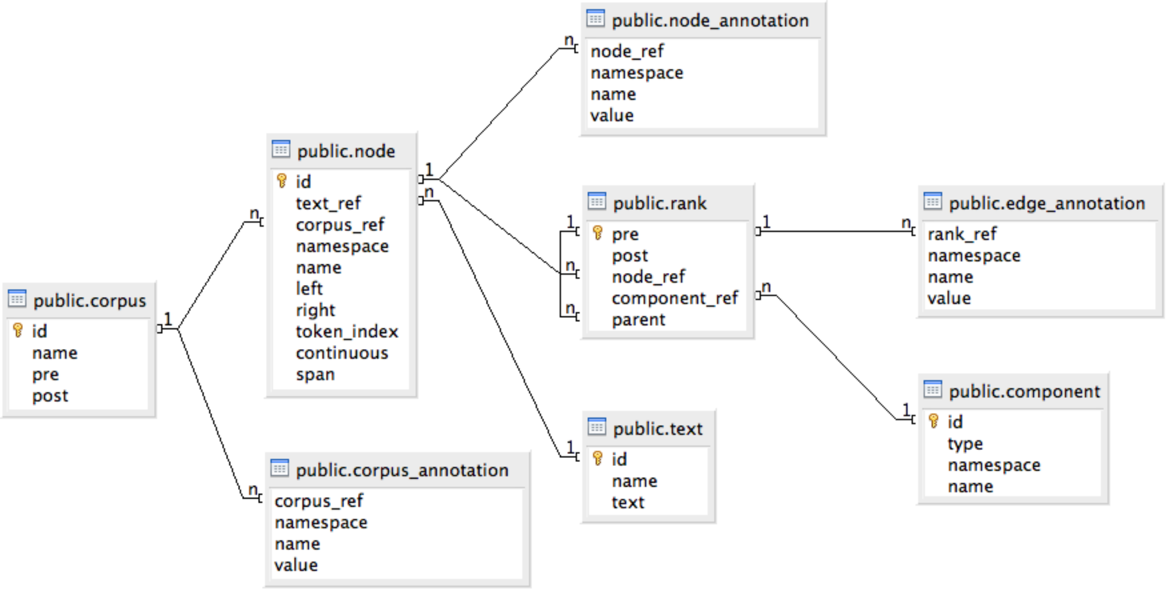


Figure 2: Relational schema of the corpus data model.

corpus_annotation:

corpus_ref	foreign key to corpus.id
namespace	optional namespace of annotation key
name	annotation key
value	annotation value

Primary data texts are stored in a table text.

text:

id	primary key
name	informational name of the primary data text
text	raw text data

Nodes of an annotation graph are stored in the tables node and node_annotation. The attributes text_ref, left, right, continuous, namespace and name of the node table correspond to the respective constituents of a node. The attribute corpus_ref is used to connect a node to a document and the attribute token_index is used to encode the token order of the underlying primary text. Finally, the attribute span contains the covered text for token spans; it could be computed from the other attributes but is supplied by the Annis converter for convenience.

node:

id	primary key
text_ref	foreign key to text.id
corpus_ref	foreign key to corpus.id
namespace	optional namespace of the node's name
name	name of the node
left	left text span border (inclusive)
right	right text span border (inclusive)
token_index	token position if the span (text_ref, left, right) is a single token, otherwise NULL
continuous	true if the span (text_ref, left, right) is gap-free, otherwise false
span	the covered text if the span is a token, otherwise NULL

node_annotation:

node_ref	foreign key to node.id
namespace	optional namespace of annotation key
name	annotation key
value	annotation value

To store the edges of the ODAG we use a combined pre/post-order scheme, originally developed as an index structure for XML documents for efficient evaluation of XPath queries [15]: Starting from a root node the graph is traversed depth-first and 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 traversal.

Because nodes can have multiple parents in an ODAG, any node except roots may be visited by the traversal algorithm more than once and thus have multiple pre/post-order values. The traversal effectively transforms an ODAG into a tree (or a forest) where nodes in different positions of the tree are identified with each other. It is, however, easy to reconstruct the original ODAG from the tree as [28] has shown.

As a consequence of this $1 : n$ relationship the pre/post-order values have to be decoupled from the nodes. They are stored in the table rank. Each row in rank represents an (incoming) edge in the transformed tree or a root node if parent is NULL. The edge type and name are not stored along with each edge; instead the annotation graph is partitioned along distinct combinations of type and name by the Annis converter and the connected components of the partitioned graph are then computed. These components are stored in the table component. Finally, edge annotations are stored in the table edge_annotation.

The attribute parent of the rank table is not strictly needed to store the ODAG as it could be computed from the other attributes. It is supplied by the Annis converter for convenience.

rank:

pre	pre-order value and primary key
post	post-order value
node_ref	foreign key to node.id
component_ref	foreign key to component.id
parent	foreign key to rank.pre of the parent node, or NULL for roots

component:

id	primary key
type	edge type of this component
namespace	optional namespace of the edges' names
name	name of the edges in this component

edge_annotation:

rank_ref	foreign key to rank.pre
namespace	optional namespace of annotation key
name	annotation key
value	annotation value

3 Annis 2 Query Language

The Annis 2 query language (AQL2) is similar to ANNIS-QL 1.0 [14], the query language used by the original Annis system which in turn is based on NiteQl [12] and TIGERSearch [19]. The most significant changes are support for edge annotations and linguistic constraints that operate on distinct graphs over the same data. Annis 2 also fixes some non-intuitive behavior of text searches and of the precedence operator in Annis 1.

3.1 Introductory example

Annis queries consist of *search terms* which select text spans by their attributes and *linguistic constraints* which specify a required relationship between the selected spans. A query can optionally contain *meta annotations* to restrict the documents that are searched by some arbitrary attribute.

An example is the easiest way to introduce these concepts:

```
1 cat="S" & node & pos="VVFIN" & node &
2 #1 >[func="OA"] #2 & #1 > #3 & #1 >[func="SB"] #4 &
3 #2 .* #3 & #3 .* #4 &
4 meta::l1="de"
```

The first line contains of four search terms. Each is assigned a number implicitly, so they can later be referred to in the query.

The next two lines describe how these four spans should relate to each other. Each line contains a number of linguistic constraints linking two spans. Line 2 states that the first span should dominate the other spans with a further restriction on the dominance relationship for the second and fourth span. Line 3 states that the second span precedes the third and that this span in turn precedes the fourth.

Finally, the last line limits the search to a subset of documents in German. Search terms, linguistic constraints and meta annotations are combined with `&` (boolean AND), and though they are shown here in order they can be mixed freely.

An answer to this query is any 4-tuple of text spans from the database that satisfies the query conditions. If searched against the PCC3 corpus this query will find sentences (`cat="S"`) in which the direct object (`func="OA"`) occurs before the verb (`pos="VVFIN"`) and the subject (`func="SB"`) after the verb. Figure 3 shows one such match.

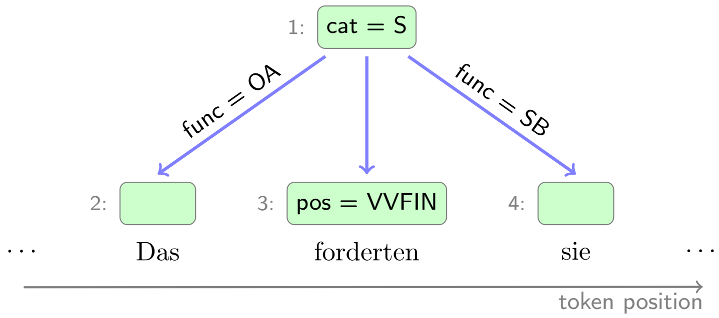


Figure 3: A match from the PCC3 corpus for the query example. Spans are represented by nodes with the covered text shown below for tokens. The number to the left of a span indicates its position in the answer tuple. Also shown are the dominance relationships between spans (blue edges) and the token order (token position axis).

A complete grammar for AQL2 is shown in appendix A.

3.2 Text span search terms

There are three ways to search for a text span:

- `node` simply selects any span in the database.
- `tiger:cat="S"` selects any span with the corresponding key-value pair annotation. The annotation key consists of a namespace and a name separated by a colon. The namespace is optional; if omitted (e.g. `cat="S"`), any annotation with the specified name will match regardless of its namespace. The annotation value can be specified using a regular expression by enclosing it with forward slashes instead of regular quotes (e.g. `cat=/S.*/`).[2] The annotation value is optional as well; if omitted (e.g. `tiger:cat`) any annotation with the specified key will match.
- `"Mary"`, or alternatively `tok="Mary"`, selects a token span that covers the corresponding text. Again, a regular expression can be used to specify the covered text (e.g. `/Mar(y|ie)/`). The search term `tok` will match any token span.

Search terms are numbered in the order they appear in the query, starting with 1.

3.3 Linguistic constraints

A linguistic constraint selects any pair of text spans that satisfy a certain relationship. The general form is `#i operator #j` where i and j are search term references. Currently four types of operators are supported: *coverage*, *precedence*, *dominance* and *pointing relations*.

Additionally, there are a few unary constraints that evaluate the properties of only one text span. These are listed in Table 6 on page 16.

3.3.1 Coverage

The coverage operation lets the user specify whether and how two text spans must overlap. They are only defined on spans of the same text, i.e. $i_{\text{text}} = j_{\text{text}}$ must hold for the spans i, j.

The definitions of all available coverage operations are listed in Table 1. Some examples are shown in Table 2.

Table 1: Coverage operations in AQL2.

Operator	**Name**	**Definition**
`#i _=_ #j`	Exact Cover	$i_{\text{left}} = j_{\text{left}} \wedge i_{\text{right}} = j_{\text{right}}$
`#i _i_ #j`	Inclusion	$i_{\text{left}} \leq j_{\text{left}} \wedge i_{\text{right}} \geq j_{\text{right}}$
`#i _l_ #j`	Left Align	$i_{\text{left}} = j_{\text{left}}$
`#i _r_ #j`	Right Align	$i_{\text{right}} = j_{\text{right}}$
`#i _ol_ #j`	Left Overlap	$i_{\text{left}} \leq j_{\text{left}} \leq i_{\text{right}} \leq j_{\text{right}}$
`#i _or_ #j`	Right Overlap	$j_{\text{left}} \leq i_{\text{left}} \leq j_{\text{right}} \leq i_{\text{right}}$
`#i _o_ #j`	Overlap	$i_{\text{left}} \leq j_{\text{right}} \wedge j_{\text{left}} \leq i_{\text{right}}$

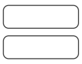
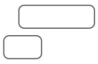
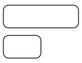
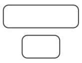
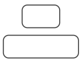
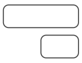
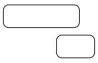
3.3.2 Dominance

The dominance operators `>` and `$` let the user specify the relative position of two spans in a syntax tree. Annis 2 allows multiple syntax trees over the same spans. These are distinguished in the model using named edges and can be queried with `>name` or `$name`. By default Annis 2 will merge all syntax trees into

---

[2] The regular expression is evaluated by PostgreSQL which uses a POSIX-style syntax with a few extensions [24]. See the PostgreSQL manual, section 9.7.3. POSIX Regular Expressions.

Table 2: Possible coverage relationships between spans (not exhaustive).

Span positions	Coverage relation						
	`#1 _=_ #2`	`#1 _i_ #2`	`#1 _l_ #2`	`#1 _r_ #2`	`#1 _ol_ #2`	`#1 _or_ #2`	`#1 _o_ #2`
1: 2: 	✓	✓	✓	✓	✓	✓	✓
1: 2: 						✓	✓
1: 2: 		✓	✓			✓	✓
1: 2: 		✓					✓
1: 2: 							✓
1: 2: 		✓		✓	✓		✓
1: 2: 					✓		✓

a unified tree which is used in dominance operations if no name is given. Table 3 shows the different versions of `>` and `$`.

Table 3: Dominance operations in AQL2.

Operator	**Definition**
`#i > #j`	i directly dominates j (alias for `#i >1 #j`)
`#i >* #j`	i indirectly dominates j
`#i >n #j`	i dominates j with distance n
`#i >n,m #j`	i dominates j with distance $n \leq k \leq m$
`#i >@l #j`	j is the left-most child of i
`#i >@r #j`	j is the right-most child of i
`#i $ #j`	i and j share a parent
`#i $* #j`	i and j share an ancestor

The direct dominance operators `>`, `>@l`, `>@r` and `$` can optionally be qualified with a list of edge annotations enclosed in brackets `[` and `]`, so that Annis 2 will only select dominance edges that are appropriately annotated. Like annotation search terms, the annotation key can be qualified with a namespace and the annotation value can be omitted or given as a regular expression.

Figure 4 shows a syntax tree fragment demonstrating dominance between spans:

- The upper `cat="PP"` span directly dominates the token span `"zum"` with a label `func="AC"` (`#1 >[func="AC"] #3`).
- It also indirectly dominates the token `"die"` (`#1 >* #4` and `#1 >2 #4`).
- The token `"Ukraine"` is the right child of the lower `cat="PP"` span (`#2 >@r #5`).
- `"die"` and `"Ukraine"` are directly dominated by the same node with a label `func="NK"` on both edges (`#1 $ [func="NK"] #2`).
- Finally, `"zum"` and `"Ukraine"` share an ancestor span in the syntax tree (`#3 $* #5`).

Dominance implies coverage, e.g. if `#1 > #2` then `#1 _i_ #2` and if `#1 >@l #2` then `#1 _l_ #2`.

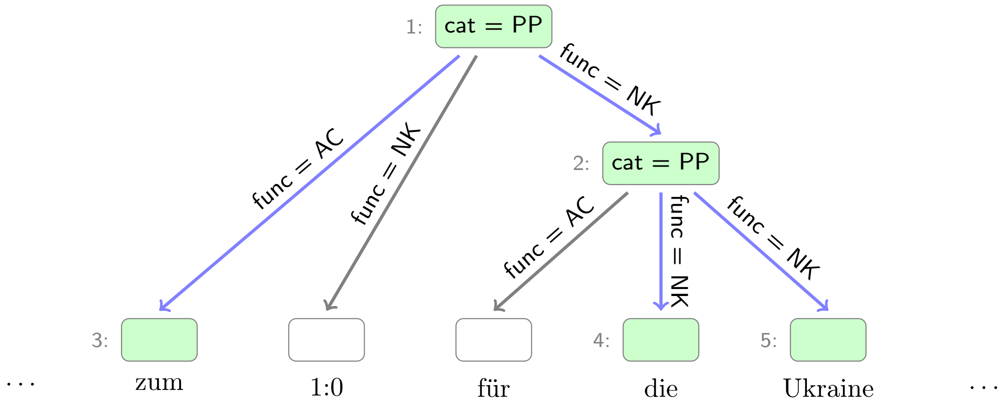


Figure 4: Syntax tree fragment demonstrating different dominance relationships between spans.

3.3.3 Precedence

The precedence operator `.` lets the user specify how many tokens two spans may be apart. Like coverage operations, precedence is only defined on spans of the same text. Table 4 shows the different versions of the precedence operator.

Table 4: Precedence operations in AQL2.

Operator	**Definition**
`#i . #j`	i directly precedes j (alias for `#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 $n \leq k \leq m$

For token spans the precedence operator reflects their order in the annotation graph. Non-token spans are not ordered in the annotation graph per se, however the order of tokens induces an order on spans in an annotation graph.

Definition 6 (Left-most, right-most covered token) *Let* s *be a span. Then* $s_{\min}$ *is the* left-most *and* $s_{\max}$ *the* right-most token covered by s.

If we assume that the token order is described by a relation $\leq_{\text{pos}}$, we can apply the precedence operator to non-token spans s and t by redefining $\leq_{\text{pos}}$ as $s \leq_{\text{pos}} t := s_{\max} \leq_{\text{pos}} t_{\min}$.

Figure 5 shows an annotation graph fragment demonstrating precedence between spans:

- The token `"zum"` directly precedes the token `"1:0"` (`#1 . #2`).
- `"zum"` also indirectly precedes the span annotated with `cat="PP"` (`#1 .* #3` and `#1 .2 #3`). Note that the token `"für"` is the left-most child of `cat="PP"` in the embedded syntax tree. Because dominance implies coverage, `"für"` is therefore the left-most token covered by `cat="PP"`.

3.3.4 Pointing relations

The pointing relation operator `->` allows queries for arbitrary links between two text spans. It follows the form of the dominance operator `>` except that the name is mandatory and that there is no support to query the left-most[3] or right-most child. Table 5 shows the different variations of `->`.

---

[3] "Left child" or "right child" would only make sense if there were a natural order on outgoing links.

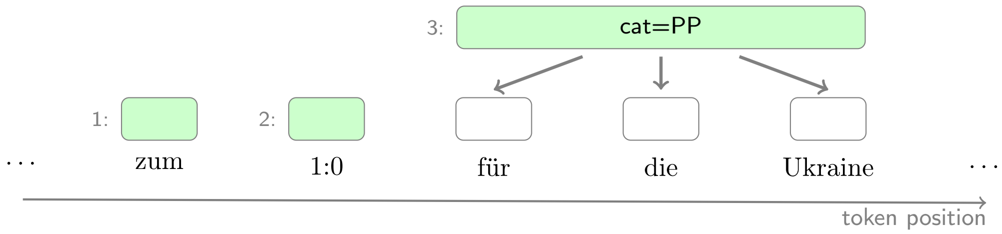


Figure 5: Annotation graph fragment demonstrating different precedence relationships between spans.

Table 5: Pointing relation operations in AQL2.

Operator	**Definition**
`#i ->name #j`	i directly points to j
`#i ->name * #j`	i points to j, either directly or through intermediate nodes
`#i ->name n #j`	i points to j with distance n
`#i ->name n,m #j`	i points to j with distance $n \leq k \leq m$

Like `>`, the direct pointing relation operator `->` can be qualified with a list of edge annotations enclosed in brackets `[` and `]`.

In Figure 6 pointing relations are used to encode the information structure of a text:

- The red link relates the pronoun *He* to *Sasha Muniak* which occurred previously in the text.
- The blue link indicates that the phrase *Polish American* further describes *Sasha Muniak*.

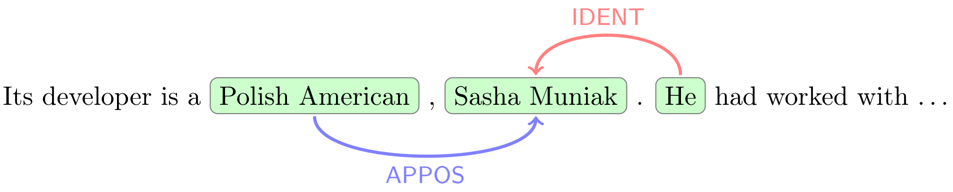


Figure 6: Using pointing relations to model information structure.

3.3.5 Text span constraints

There are a few unary linguistic constraints that only operate on one search term. They are listed in Table 6.

3.4 Combining expressions with OR

The introductory example has already shown how search terms and linguistic constraints are combined with AND to build non-trivial queries. They can also be grouped with parentheses `(` and `)` and combined with `|` (logical OR) to build more complex expressions such as the following query:

```
"the" & (("tree" & #1 . #2) | ("house" & #1 . #3))
```

This query could be stated in a more concise fashion using a regular expression search:

```
"the" & /tree|house/ & #1 . #2
```

However, the longer version using OR is much faster; see section 6.7 for details.

Table 6: Unary linguistic constraints in AQL2.

Operator	**Definition**
`#i:root`	i is a root node of an annotation graph
`#i:arity = n`	i has n children
`#i:arity = m,n`	i has $m \leq k \leq n$ children
`#i:tokenarity = n`	i covers n token
`#i:tokenarity = m,n`	i covers $m \leq k \leq m$ token

3.5 Meta data

Given a query, Annis 2 generally searches all documents of a corpus, but the search can be confined to documents that have a specified meta annotation. The general form is `meta::namespace:key="value"` which looks like an annotation search term prepended with `meta::`. As with annotation search terms, the namespace and the value are optional and the value can be given as a regular expression. Note that although the syntax is similar, meta annotation definitions do not count as search terms and are skipped when evaluating search term references in linguistic expressions. 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.

3.6 Query evaluation

Annis 2 evaluates queries in the following fashion: Let q be an AQL2 query and C a set of corpora on which q should be evaluated.

First, if q contains meta annotation constraints they are stripped from q. Let D be the set of documents in C that satisfy every meta annotation constraint originally contained in q (or the set of all documents in C if q did not originally contain any meta annotation constraints).

Then q is transformed into its disjunctive normal form $q' = q_1 \vee \ldots \vee q_n$ and each alternative is checked for validity.

Definition 7 (Valid query) *Let q be a query, q_i an alternative of the disjunctive normal form of q and let q_i contain $k \geq 2$ search terms $s_1, \ldots, s_k$ and $l \geq 1$ binary linguistic constraints $c_1, \ldots, c_l$. Further, let $\#r$ be the reference to the r-th search term in q_i, and let $\otimes$ be a placeholder for an arbitrary binary linguistic operator. Then q_i is* valid *iff*

$$\begin{array}{ll} \forall s_i, s_j : & (\exists c_p : c_p \text{ is of the form } \#i \otimes \#j) \\ & \vee \\ & \left(\begin{array}{c} \exists s_{r_1}, \ldots, s_{r_n}, c_{r_1}, \ldots, c_{r_{n+1}} : \\ c_{r_1} \text{ is of the form } \#i \otimes \#r_1 \text{ or } \#r_1 \otimes \#i \wedge \\ c_{r_2} \text{ is of the form } \#r_1 \otimes \#r_2 \text{ or } \#r_2 \otimes \#r_1 \wedge \\ \ldots \\ c_{r_n} \text{ is of the form } \#r_{n-1} \otimes \#r_n \text{ or } \#r_n \otimes \#r_{n-1} \wedge \\ c_{r_{n+1}} \text{ is of the form } \#r_n \otimes \#j \text{ or } \#j \otimes \#r_n \end{array} \right) \end{array}$$

A query alternative consisting of exactly one search term and no linguistic constraints is always valid. The query q is valid *iff all of its alternatives are valid.*

Informally, if we consider the search terms of an alternative as the nodes and the binary linguistic constraints as the (undirected) edges of a graph then the alternative is valid iff the corresponding graph is connected.[4]

---

[4] This requirement is necessary because the behavior of Annis 1, which implicitly assumes that unconnected groups of

Finally, for each alternative q_i and each document $d \in D$ Annis 2 will try to assign a span from d to each of the k span selection terms, so that each of the l linguistic constraints is satisfied.

Definition 8 (Satisfied constraint, Solution) *Let q be a query, q' its disjunctive normal form, and q_i an alternative of q' with k search terms and l linguistic constraints $c_1, \ldots, c_l$ and let c_j be of the form #s $\otimes$ #t with $1 \le s, t \le k$ and $\otimes$ is a binary*[5] *linguistic operator. Further, let $d \in D$ be a document and T be a k-tuple of spans from d. We identify #s and #t with the s-th and t-th component of T respectively. Then T* satisfies *c_j, iff the spans T_s and T_t are in the relationship described by $\otimes$. We call T a* solution for *q_i, iff T satisfies every c_j in q_i. We call T a* solution for *q, iff there exists an alternative q_i of q' for which T is a solution.*

Note that each alternative in q' may have a different number of span selections terms. Therefore, the size of a solution for q is not fixed, if its disjunctive normal form q' consists of more than one alternative. Annis 2 does not identify the alternative of which a tuple T is a solution but such identification is possible if the annotation graph fragment over T is retrieved. This process is described in the next section.

3.7 Query functions

Knowing which spans are a solution to an Annis query is not all that interesting in itself. Researchers are typically interested in context or aggregate information. To this end, Annis 2 defines query functions that evaluate a query against a set of corpora and then retrieve additional information from the database based on the solutions to the query.

Strictly speaking, query functions are not part of the Annis 2 query language. Instead, they encapsulate the steps that have to be taken to further analyze the solutions to a query after the solutions have been computed. Each query function corresponds to a analysis strategy supported by the Annis web interface.

Before we can discuss query functions we have to define a few more concepts.

Definition 9 (Preceding, following token) *Let t be a token and let $n \in \mathbb{N}$. Then $t - n$ is the token preceding t by n tokens and $t + n$ is the token following t by n tokens.*

Definition 10 (Annotation graph fragment) *Let T be a set of primary data texts, $A = (V, E)$ an annotation graph over T with nodes V and edges E and S a solution to a query from A in T. An* annotation graph fragment over S with left context l and right context r *is the subgraph of A consisting of the node set*

$$V' = \bigcup_{s \in S} \{v \in V : v \text{ overlaps a token from the interval } [s_{\min} - l, s_{\max} + r]\}$$

and the edge set

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

Informally an annotation graph fragment contains all the tokens that are at most l tokens to the left or r tokens to the right of a span in a query solution, any span that overlaps these tokens, and all the edges in-between them.

Definition 11 (Annotation matrix) *Let A be an annotation graph, q an AQL2 query, S the set of solutions to q in A and let m be the maximum number of spans in a solution in S. The* annotation matrix of S *is a matrix that is constructed in the following fashion:*

---

spans overlap, is very costly to implement. A query with n search terms and no linguistic constraint would require the addition of 2^n alternative overlap constraints. The overlap operation is costly in itself (see section 6.1); to emulate the behavior of Annis 1 is thus prohibitive.

[5] The process for unary operators is analogous: T satisfies an unary linguistic constraint of the form #s:$\otimes$, iff the span T_s has the property described by $\otimes$.

1. *For each tuple position $1 \leq p \leq m$, the annotation keys*[6] K_p *of any span* T_p *at position* p *in a solution* $T \in S$ *are determined.*

$$K_p = \bigcup_{T \in S} \{k : k \text{ is an annotation key of the } p\text{-th span in } T\} .$$

2. *The header of the matrix, i.e. the first row, is constructed by creating* $\sum_{p=1}^{m} \|K_p\|$ *cells, one cell for each tuple* (p, k) *with* $k \in K_p$.
3. *The body of the matrix, consisting of the following* $\|S\|$ *rows, is constructed by creating a row for each solution* $T \in S$. *If the span* T_p *is annotated with a key* $k \in K_p$ *then the corresponding cell contains the annotation value, otherwise it is empty.*

Currently Annis 2 defines the following three query functions. Let q be an AQL2 query, C a set of corpora on which q should be performed and S the set of solutions to q in C.

- $COUNT\,(q, C)$ returns the number of solutions in S.
- $ANNOTATE\,(q, C, l, r)$ returns for each solution $s \in S$ the annotation graph fragment over s with l tokens as left context and r tokens as right context.
- $MATRIX(q, C)$ returns an annotation matrix for S in ARFF-Format [1].

3.8 Pagination of $ANNOTATE$ results

The results returned by the $ANNOTATE$ query function can quickly become very large because it returns a complete annotation graph fragment for each tuple of spans matching the underlying query. Not only is the height of the annotation graph fragment unknown, but the width of the fragment can be arbitrarily extended by a user-defined context. Since the $ANNOTATE$ function is designed to be used interactively, returning the annotation graph fragments for every result is not sensible as the amount of the information presented would easily overwhelm the user. The frontend therefore enforces a pagination of the $ANNOTATE$ results similar to a web search engine: only the first n results of the query are displayed and the user can retrieve the next results if wanted. The number of results displayed on a page is user-configurable.

Note that the Annis service supports the retrieval of all $ANNOTATE$ results at once; however, as section 6.6 shows, it is not optimized for this use case (the database handles this use case just fine).

3.9 Differences between ANNIS-QL 1 and AQL2

Although queries of the two languages look similar, there are quite a few differences:

- A text search only covers tokens. There is currently no possibility to perform a real full text search in Annis 2. For example, in Annis 1 one can search for the phrase `"the house"`; in Annis 2 each token has to be specified separately and linked with the precedence operator: `"the" & "house" & #1 . #2`.
- A text search has to match the entire text covered by a token. In Annis 1 a text search would implicitly match substrings as well. To match a substring in Annis 2 one can use a regular expression.
- In Annis 1 one could search for spans by defining an annotation and the covered text in one expression (`key=value:"text"`). This syntax was very confusing in practice and is no longer allowed. The same search can be achieved with `key="value" & "text" & #1 _=_ #2`.

---

[6] For the purpose of this definition, the covered text of a span is considered an annotation of the span with the key *span*.

- Annis 1 evaluates precedence in terms of the left and right text border of spans which results in non-intuitive behavior. For example, to search for an adjective followed by the string *tree*, one would have to write `pos=ADJ & " tree" & #1 . #2`. Note the space in `" tree"` which assumes that that all tokens in the text are separated by exactly one space. The Annis 2 corpus model makes the tokenisation that is present in the original data explicit and evaluates precedence in terms of the token position. Thus, in Annis 2 the query can be written as expected: `pos="ADJ" and "tree" & #1 . #2`.
- There is no document search (`doc=maz.*`) in Annis 2. Using meta data to select documents is a much more powerful alternative.
- Annis 1 implicitly assumes that text spans that are not used in any linguistic expression have to overlap. For example, `pos=VVINF & cat=S` would be converted to `pos=VVINF & cat=S & #1 _o_ #2` before evaluation. In Annis 2, the first form is no longer possible because all search terms have to be connected to each other by a linguistic operation directly or indirectly.
- Annis 1 interprets a single identifier as either a key or a value, e.g. `pos` would be expanded to `pos=* | *=pos`. Annis 2 treats single identifiers as an existence query, i.e. it selects any span annotated with the corresponding key, regardless of the annotation value.
- Annotation type sets are not supported by Annis 2.
- Annis 2 does not yet support NOT or XOR.
- Annis 2 only allows normal parentheses `(` and `)` to group expressions. Brackets `[` and `]` are used to define edge labels.

4 SQL Generation

In this section we will describe how an AQL2 query is translated into a SQL query. For historical reasons an Annis query is not translated directly to SQL, but translated to an intermediate DDD*query* [29] first. The mapping from AQL2 language features to DDD*query* language features is described in appendix B. Since it is almost trivial in nature, the rest of this section skips this intermediary step and assumes that the Annis query is translated directly to SQL. The translation from AQL2 to DDD*query* is briefly described in appendix B.

We will first demonstrate how to build a SQL query that generates all the solutions for a given Annis query from a list of corpora. Then, we will extend this general framework to include query functions as described in section 3.7.

4.1 Computation of derived node data during corpus import

The evaluation of some operations requires information that is not explicitly present in the corpus schema and which first has to be derived from other data contained therein. This information is fixed for each node; it is therefore advisable to perform the computation only once during corpus import and cache the results in the node and rank tables.

However, each additional column will generally slow down the evaluation of queries because for each node PostgreSQL has to load more data from disk. A trade-off has then to be found between improving the performance of a specific operation and the general performance on typical queries. For example, we have decided against caching the node arity described in section 4.3.9 because we have not seen it used in actual queries.

4.1.1 Minimally and maximally covered tokens

In section 3.3.3 we extended the token order relation $\leq_{\text{pos}}$ to non-token spans s and t by comparing the right-most and left-most covered token s_{max} and t_{min}:

$$s \leq_{\text{pos}} t := s_{\text{max}} \leq_{\text{pos}} t_{\text{min}}$$

During import we set $t_{\text{min}} := t_{\text{max}} := t$ for each token span t and compute s_{min} and s_{max} for each non-token span s. We then extend the node table with the attributes left_token and right_token which for each span s store the value of node.token_index of s_{min} and s_{max} respectively.

4.1.2 Root nodes in the original ODAG

In section 4.3.7.2 we describe how the original ODAG is partitioned to implement the dominance and pointing relationship operators. This can create partitions rooted in a node that is a leaf in another partition; in other words it creates many false roots. A true root in the original ODAG will be a root node in any partition it appears in. The following SQL query finds all such nodes:

```
SELECT node_ref
FROM rank
GROUP BY node_ref
HAVING count(DISTINCT rank.parent) = 0;
```

During import we extend the rank table with the attribute root and set it to TRUE if the corresponding node is selected by the above query and to FALSE if it is not.

4.1.3 Identification of a node's top-level corpus

The corpus schema defined in section 2.3 links each node with the document that contains the corresponding text span. If a search is restricted to a document d, all documents below d have to searched as well; in section 4.5 we describe a general way to achieve just this.

However, the Annis frontend only exposes top-level corpora and each top-level corpus is imported individually. It is therefore easier and faster to extend the node table with an attribute toplevel_corpus which is a foreign key to corpus.name and store in it the name of top-level corpus being imported.

4.2 The SELECT and FROM clauses

Internally, a span is identified by the primary key of its node in the database database, node.id. This attribute is (mostly) meaningless to the Annis frontend but the SELECT clause is highly dependent on the query function used, and right now we are only interested in generating solutions for a given query.

Consider a query without disjunctions, containing n search terms. For this query we access the node table via n aliases in the FROM clause and select their id attributes in the SELECT clause. This strategy generates one row in the result set for each solution to the query. We use the DISTINCT keyword to ensure set semantics.

```
SELECT DISTINCT
  node1.id, node2.id, ..., nodeN.id
FROM
  node AS node1, node AS node2, ..., node AS nodeN
...
```

Recall that the tuple length is not necessarily the same for each solution if the query contains more than one alternative. However, in the SQL fragment above we have fixed the number of columns and accessed table aliases. We will defer the resolution of this conflict to section 4.4 and assume for the rest of this section that the Annis query consists of only one alternative, unless otherwise noted.

The SELECT clause is now complete. The FROM clause on the other hand only reference the node table which contains the necessary information to implement a node or text search and the precedence and coverage operators. If the query contains an annotation search or any other linguistic constraint, the tables node_annotation, rank, component and edge_annotation are needed.

If a query requires information from another table to implement an operation involving a search term, the compiler will create a table alias for this table and join it to the corresponding node table alias. It is sufficient to join each table only once to the node alias except for the edge_annotation table. The direct dominance and pointing relation operators can be qualified with multiple edge labels and the compiler has to create one edge_annotation table alias for every label.

Listing 1 shows the FROM clause that is generated for the query example in section 3.1.

4.3 The WHERE clause: Translation of AQL2 language features

In this section we will show how AQL2 language features translate to conditions in the WHERE clause. For search terms and unary linguistic constraints we will assume that the appropriate tables are accessed via an alias with index 1 in the FROM clause. For binary constraints we assume the existence of table aliases with index 1 for the left-hand-side span and index 2 for the right-hand-side span.

4.3.1 Text search

A text search `"Mary"` or `tok="Mary"` is realized by comparing the text with the attribute node1.span.

Listing 1: FROM clause generated for the Annis query example in section 3.1.

```
FROM
 node AS node1
  JOIN node_annotation AS node_annotation1 ON (node_annotation1.node_ref = node1.id)
  JOIN rank AS rank1 ON (rank1.node_ref = node1.id)
  JOIN component AS component1 ON (rank1.component_ref = component1.id),
 node AS node2
  JOIN rank AS rank2 ON (rank2.node_ref = node2.id)
  JOIN component AS component2 ON (rank2.component_ref = component2.id)
  JOIN edge_annotation AS edge_annotation2_1 ON (edge_annotation2_1.rank_ref = rank2.pre),
 node AS node3
  JOIN node_annotation AS node_annotation3 ON (node_annotation3.node_ref = node3.id)
  JOIN rank AS rank3 ON (rank3.node_ref = node3.id)
  JOIN component AS component3 ON (rank3.component_ref = component3.id),
 node AS node4
  JOIN rank AS rank4 ON (rank4.node_ref = node4.id)
  JOIN component AS component4 ON (rank4.component_ref = component4.id)
  JOIN edge_annotation AS edge_annotation4_1 ON (edge_annotation4_1.rank_ref = rank4.pre)
```

```
node1.span = 'Mary'
```

Note that the `span` attribute contains the content of a text span for tokens only and is set to NULL for non-tokens. This corresponds to the property of the text search that it can only be used for token spans.

To implement a text search with a regular expression like `/Mar(y|ie)/` we use the PostgreSQL-specific operator ~ which matches its left-hand side to a regular expression on the right-hand side.

```
node1.span ~ '^Mar(y|ie)$'
```

Note that the regular expression is always explicitly anchored as required by the definition of the regular expression text search in AQL2.

4.3.2 Token search

To search for any token we simply select those tuples from the `node` table where the attribute `span` is not set to NULL:

```
node1.span IS NOT NULL
```

Alternatively, we can use the `token_index` attribute which is also set to NULL for any non-token span:

```
node1.token_index IS NOT NULL
```

4.3.3 Annotation search

To implement an annotation search we compare the components of the search term to the attributes `namespace`, `name` and `value` of the `node_annotation` table. For example, a search for a span annotated with `tiger:cat="S"` is realized by the following conditions:

```
node_annotation1.namespace = 'tiger' AND
node_annotation1.name = 'cat' AND
node_annotation1.value = 'S'
```

An annotation search with a regular expression is implemented in the same manner as a regular expression text search; the regular expression is explicitly anchored and matched using the PostgreSQL pattern matching operator `~` (tilde). If an optional part of an annotation search is missing then the constraint on the corresponding table attribute is omitted as well.

4.3.4 Node search

The search term `node` places no constraint on the spans selected from the annotation graph. Accordingly, it requires no conditions in the WHERE clause. Simply listing an alias to the node table in the FROM clause is enough to implement a node search.

4.3.5 Coverage

Coverage operations are defined as a comparison of the left and right borders of two spans of the same text. The span borders are stored in the attributes node.left and node.right and the text is referenced by the foreign key node.text_ref. To implement a coverage operator, we substitute each span property in the operator definitions in Table 1 in section 3.3.1 with the corresponding table attribute of the node table (see section 2.3).

For example, the exact-cover constraint `#i _=_ #j` is realized by the following conditions:

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

4.3.6 Precedence

In Annis 2, precedence is defined in terms of tokens. The term `#i . #j` conveys that there should be no token between the spans i and j, regardless of any possible whitespace that may exist between them in the original primary text.[7] Similarly, the term `#i .n #j` conveys that i and j should be exactly n tokens apart (which is not possible to express in Annis 1 at all).

In section 3.3.3 we have shown how the token order relation $\leq_{\text{pos}}$ is extended to non-token spans s and t by comparing the right-most and left-most covered token s_{max} and t_{min}:

$$s \leq_{\text{pos}} t := s_{\text{max}} \leq_{\text{pos}} t_{\text{min}} \tag{1}$$

We can use Equation 1 to formally define the variants of the precedence operator listed in Table 4:

$$\begin{array}{lcl} \texttt{\#i . \#j} & \iff & i_{\text{max}} = j_{\text{min}} - 1 \\ \texttt{\#i .* \#j} & \iff & i < j \\ \texttt{\#i .n \#j} & \iff & i_{\text{max}} = j_{\text{min}} - n \\ \texttt{\#i .n,m \#j} & \iff & j_{\text{min}} - m \leq i_{\text{max}} \leq j_{\text{min}} - n \end{array} \tag{2}$$

The right-hand side of the defintions in Equation 2 can be translated to SQL using the attributes left_token and right_token of the appropriate node table which were computed during corpus import. Additionally, the comparison has to be restricted to spans of the same text. For example, the term `#i . #j` is translated as follows:

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

---

[7] This is substantially different from Annis 1; see section 3.9 for more information.

4.3.7 Dominance and pointing relations

Both dominance and pointing relations between two spans are modelled as an edge (or a path) between the corresponding nodes in the annotation graph. The position of a node in a graph is in turn encoded by (combined) pre- and post-order values, which are stored in the attributes `pre` and `post` of the `rank` table.

We use the annotation graph shown in Figure 7 as a running example while discussing the implementation of the dominance and pointing relation operators.

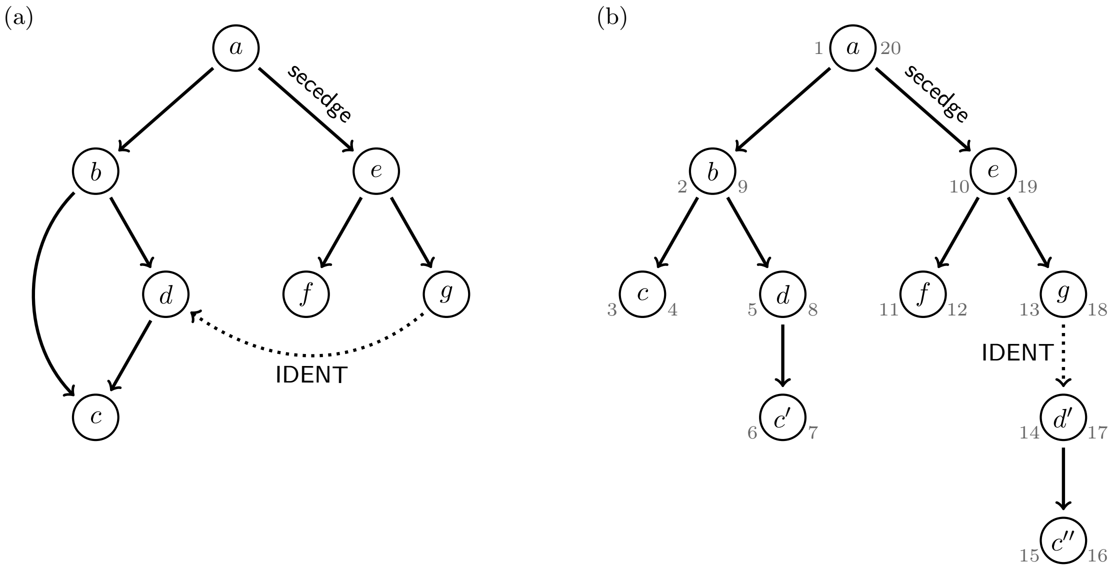


Figure 7: Annotation graph with pointing relations and multiple syntax trees; (a) the original ODAG and (b) the equivalent decomposed tree as seen by the pre/post-order traversal. Solid edges denote a dominance relation named *edge* unless labeled otherwise.[8] Dotted edges denote pointing relations. A node v' in the decomposed tree denotes a copy of a previously encountered node v with different pre- and post-order values.

4.3.7.1 Basic strategy

For each node in a tree the pre- and post-order values partition the tree into four distinctive regions which correspond to the `ancestor`, `descendant`, `preceding` and `following` axis in XPath [15]. To implement dominance (and pointing relations) we only need to test for nodes along the `descendant` axis:

$$\begin{aligned} i \text{ dominates } j &\iff j \text{ is a descendant of } i \\ &\iff i_{\text{pre}} < j_{\text{pre}} \wedge i_{\text{post}} > j_{\text{post}} \\ &\iff i_{\text{pre}} < j_{\text{pre}} < i_{\text{post}} \end{aligned} \tag{3}$$

The last transformation in Equation 3 is motivated by the usage of one counter for both pre-order and post-order. If we substitute the corresponding table attributes we arrive at:

```
rank1.pre < rank2.pre AND
rank2.pre < rank1.post
```

Of course, for direct dominance we can exploit the attribute `rank.parent` which is conveniently provided by the Annis converter:

```
rank1.pre = rank2.parent
```

---

[8] Dominance edges called *edge* and *secedge* are artifacts of a TIGERSearch-annotated corpus.

4.3.7.2 Separation of dominance and pointing relation edges in the annotation graph

However, the definition of the dominance relation in Equation 3 is incomplete, since it tests for the existence of *any* path between two spans and disregards the semantic meaning of the edges along that path. For example, in Figure 7 the span g does not dominate d' because the edge between g and d' encodes a pointing relation. Even if the first and the last edge of a path are dominance edges, there might still be a pointing relation edge somewhere in the middle such as in the path from e to c''. It is therefore necessary to restrict Equation 3 in such a way that all edges along a path between two spans are of the same type.

To this end, we partition the annotation graph into connected components such that each edge in a particular component is of the same type. The implementation of the dominance relation can then be completed by checking the edge type of the first (or last) edge and making sure that there is a component that contains both spans. This can be expressed with the following conditions:

```
component1.type = 'd' AND
component1.id = component2.id
```

The Annis converter computes the pre- and post-order values in such a way that Equation 3 holds, iff there exists a component with both spans. The last condition can thus be omitted.

Figure 8 shows the components of the annotation graph in Figure 7. As expected there is no component of dominance edges that contains a path from g to d' or from e to c''.

If the original annotation graph contains nodes that are connected to two or more parent nodes by different edge types (e.g. the span d), then the partitioning strategy will create components that are completely contained in another component in the partitioned graph (e.g. component (c) is contained in (a) in Figure 8). These are pruned from the database by the Annis converter to save space.

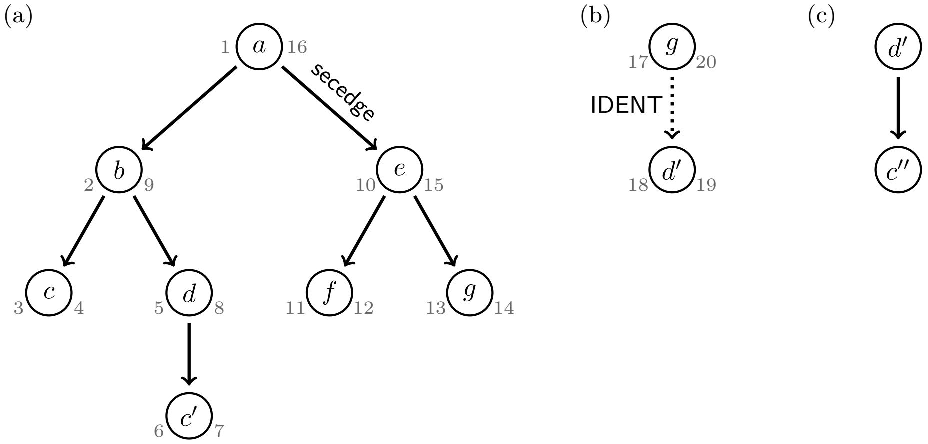


Figure 8: Annotation graph partitioned by edge type. The components (a) and (c) contain only dominance edges while (b) contains only pointing relation edges. Component (c) is pruned from the database.

Edges in Annis have a second property that conveys semantic meaning, their name. While named edges are rarely used in dominance relations, they are mandatory for pointing relations: the term `#i ->IDENT * #j` requires that *all* edges on the path from i to j have the name IDENT.

Conceptionally, a name is nothing but a subtype. To ensure that all edges along a path have the same name, we partition the graph along the distinct combinations of edge type and name. Figure 9 shows the generated components for the annotation graph in Figure 7. We can now ensure that every edge along a path has a certain name, by checking the name of the first (or last) edge:

```
component1.name = 'IDENT'
```

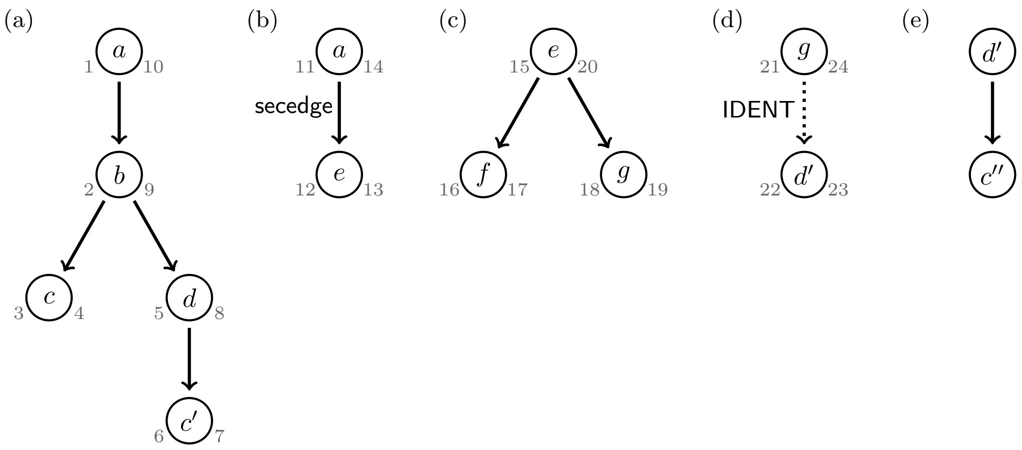


Figure 9: Annotation graph partitioned by edge type and name. Component (e) is pruned from the database because it is contained in component (a).

4.3.7.3 The unnamed dominance operator variant

Annis 2 will merge all syntax trees of a given graph into one unified tree that is searched if the dominance operator is used without a name. Consequently, the term `#i >* #j` only requires that edges on a path from i to j are dominance edges. Their name is not only irrelevant but each edge can have a different name as in the path from a to g in Figure 7. Unfortunately, when we partitioned the annotation graph along edge names, we only retained paths where each edge has the same name.

The solution is to take the dominance components of the type-partitioned graph, set their name to NULL and test for a path in any of these components. Figure 10 shows all the components that are created for the annotation graph in Figure 7.

4.3.7.4 Length-restricted dominance operator variants

To implement the length-restricted variants of the dominance operator `>n` and `>n,m`, we need to be able to measure the length of a path between two nodes. We can achieve this by computing the depth of each node and only return those nodes i and j which are connected by a path of length

$$n = j_{\text{depth}} - i_{\text{depth}}. \tag{4}$$

At first glance, this approach poses three problems:

1. The depth of a node is defined as the distance from the root to the node. An annotation graph can have multiple roots; which one should we choose?
2. For any two nodes i and j there may be multiple paths between i and j in the annotation graph, each with a different length.
3. The partitioning described in section 4.3.7.2 can introduce many false roots, such as the node e in Figure 10. As a consequence, the computation of the node depth will yield different results depending on whether it is done before or after graph partitioning.

The first issue disappears once we have assigned pre- and post-order values to the nodes. If the original annotation graph had multiple roots, the pre/post-order traversal generates a forest of trees and we can compute the node depth for each tree individually.

The second issue is actually a feature of Annis. For example, we want to find the spans b and c as solutions for the term `#i > #j` as well as `#i >1 #j`. This is possible because the pre/post-order traversal effectively creates a copy of c to which we can assign a different depth.

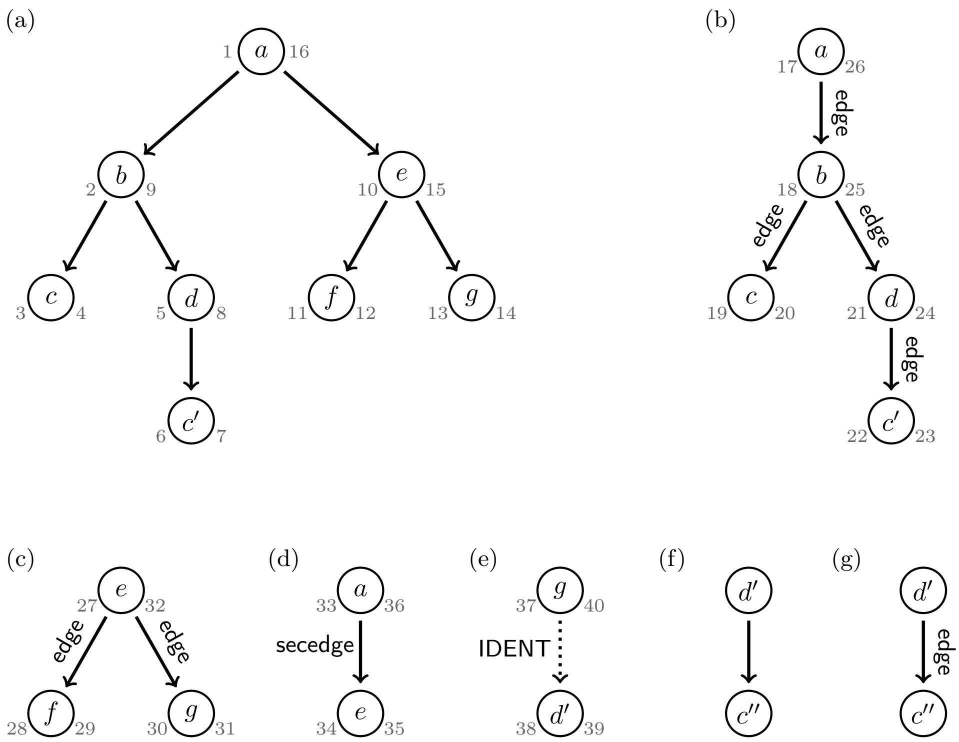


Figure 10: Annotation graph components for each edge type and name and the merged syntax tree. Components (a) and (f) contain the merged syntax tree, components (b), (c) and (g) contain dominance edges called *edge*, component (d) contains a *secedge* dominance edge and component (e) an IDENT pointing relation. Components (f) and (g) are pruned from the database because they are contained in (a) and (b) respectively.

Finally, the exact values for the node depth are irrelevant, since we're only interested in the distance between nodes.

It follows from these considerations that the depth of a node has to be stored in the `rank` table. During import we extend the `rank` table with the attribute `depth` in which we store the depth for each node. Substituting this attribute into Equation 4 we arrive at the following condition for the term `#i >n #j`:

```
rank1.depth = rank2.depth - n
```

The term `#i >n,m #j` is implemented in a similar fashion:

```
rank1.depth BETWEEN SYMMETRIC rank2.depth - n AND rank2.depth - m
```

4.3.7.5 Left-most and right-most dominance operator variants

To implement the dominance operators `>@l` and `>@r` we exploit the fact that the computation of the pre- and post-order values follows the order of the children of a node. In other words, for each non-terminal i and its left-most child l and right-most child r the following conditions hold:

$$\begin{aligned} i_{\text{pre}} &= l_{\text{pre}} - 1 \\ i_{\text{post}} &= r_{\text{post}} + 1 \end{aligned} \tag{5}$$

In the original unpartitioned annotation graph the left-most (or right-most) child identified by Equation 5 could potentially be connected to the parent node by a pointing relation edge. However, once the annotation graph is partitioned along edge types, there can be no pointing relation edge in a dominance component. Substituting the corresponding table attributes into Equation 5 we arrive at the following condition for `>@l`:

```
rank1.pre = rank2.pre - 1
```

`>@r` is implemented in a similar fashion:

```
rank1.post = rank2.post + 1
```

4.3.7.6 Same parent and same ancestor dominance operator variants

The implementation of the dominance operator `$` is simple as we explicitly store a pointer to the parent of a node in the attribute `rank.parent`. Thus, two nodes share a parent if there exists a dominance component for which the corresponding entries in the `rank` table have the same value in `parent`:

```
rank1.parent = rank2.parent
```

If two nodes s and t share a common ancestor c then the following Equation 6 must hold for both c and s and c and t:

$$\begin{aligned} s \text{ and } t \text{ share an ancestor} &\iff \exists \text{ node } c : c_{\text{pre}} < s_{\text{pre}} < c_{\text{post}} \land c_{\text{pre}} < t_{\text{pre}} < c_{\text{post}} \\ &\implies s,\ t \text{ and } c \text{ are connected} \end{aligned} \tag{6}$$

We can express this relationship using a subquery in an EXISTS clause:

```
component1.id = component2.id AND
EXISTS (
  SELECT 1 FROM rank AS ancestor WHERE
  ancestor.component_ref = component1.id AND
  ancestor.pre < rank1.pre AND rank1.pre < ancestor.post AND
  ancestor.pre < rank2.pre AND rank2.pre < ancestor.post
)
```

4.3.7.7 Edge annotations

Dominance and pointing relation operations between parent and child spans (i.e. `>`, `>@l`, `>@r`, `$` and `->`) may be qualified with a list of edge annotations to search for particular edges. The test for an edge annotation is similar to a node annotation search using the `edge_annotation` table which contains a foreign-key to the `rank` table. Because each tuple in `rank` is interpreted as an *incoming* edge, we have to use the `edge_annotation` table alias created for the right-hand-side of the operator which is expressed by the index 2 *before* the underscore.

```
edge_annotation2_1.namespace = 'tiger' AND
edge_annotation2_1.name = 'func' AND
edge_annotation2_1.value = 'OA'
```

Recall that we can qualify the dominance and pointing relation operator with multiple edge annotations. We need to access a different `edge_annotation` table alias for each listed annotation. In the example above we have assumed that we are looking for the first annotation in the list, denoted by the index 1 *after* the underscore.

The same parent operator `$` is an exception to the rule that we have to test the right-hand-side of the operator. For `$` we want to make sure that both nodes are connected to a parent by accordingly annotated edges. We thus need to test the `edge_annotation` aliases created for both nodes.

4.3.8 Root nodes

In the original unpartitioned annotation graph a root node is identified by its parent attribute being set to NULL. Unfortunately, the partitioning of the annotation graph described in section 4.3.7.2 may introduce many false roots, i.e. nodes which are a root node in one component and a leaf node in another component. For example, in Figure 10 the node e is the root of component (c) and a leaf in component (d).

To correctly implement the root operator `#i:root`, we must only consider nodes as roots which are a root node in all components in which they appear. These are identified by the attribute root of the rank table which is computed during corpus import. The term `#i:root` can thus be implemented by testing the root attribute:

```
rank1.root IS TRUE
```

4.3.9 Node arity

The node arity, i.e. the number of children for a given node v, can be determined by counting the entries in the rank table which point to v via the parent attribute. The term `#i:arity=n` can thus be implemented as follows:

```
(SELECT count(DISTINCT children.pre)
 FROM rank AS children
 WHERE children.parent = rank1.pre) = n
```

The term `#i:arity=n,m` is implemented in a similar fashion:

```
(SELECT count(DISTINCT children.pre)
 FROM rank AS children
 WHERE children.parent = rank1.pre) BETWEEN SYMMETRIC n AND m
```

We have left out the restriction on a particular component in the SQL fragments above because we assume that it is selected by another linguistic constraint in the Annis query.

4.3.10 Token arity

During corpus import we have identified the left-most and right-most covered token of a given span s and stored their index in the attributes node.left_token and node.right_token respectively. We can use these attributes to implement the token arity operator:

$$s \text{ covers } n \text{ token} \iff n = s_{\max} - s_{\min} + 1 \tag{7}$$

Transforming Equation 7 and substituting the corresponding table attributes, we arrive at the following conditions for the term `#i:tokenarity=n`:

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

Similarly, the term `#i:tokenarity=n,m` can be translated as follows:

```
node1.text_ref = node2.text_ref AND
node1.left_token BETWEEN SYMMETRIC
      node2.right_token - n + 1 AND node2.right_token - m + 1
```

This concludes the SQL generation for a query with only one alternative. In the next section, we will see how to transform queries that contain multiple query alternatives into SQL code.

4.4 Query alternatives

During the discussion of the SELECT and FROM clauses we have already alluded to an apparent shortcoming of the strategy to model a query solution as a tuple of node.id attributes: whereas the solutions to a query with multiple alternatives may vary in size, the tuple size returned by a SELECT statement is necessarily fixed.

Suppose that we have a query q consisting of two alternatives q_1 containing n search terms and q_2 containing m search terms and $n < m$. We will then need at least m aliases for the node table and any other table that may be required to solve the alternative q_2. Suppose further that we use the first n table aliases for both alternatives and construct the query for q in the following fashion:

```
SELECT DISTINCT
      node1.id, node2.id, ..., nodeM.id
FROM
      node AS node1 JOIN ...,
      node AS node2 JOIN ...,
      ...,
      node AS nodeM JOIN ...
WHERE
      ( conditions for q1 ) OR
      ( conditions for q2 )
```

This strategy causes two interrelated problems:

1. Let's assume that we need to join an annotation table to the node table representing the span at position p in the solutions for q_1. Let's assume further, that there exists no tuple in this annotation table that could be joined against the nodes selected at position p for q_2.[9] Because of the semantics of JOIN, an empty set will be selected for the candidates at position p which in turn will produce an empty result for the entire alternative q_2.
2. Because the conditions generated for q_1 place no constraint on the table aliases with an index bigger than n, the solutions returned for q_1 will consist of the cartesian product of the actual solutions and any span in the database for the tuple positions $n + 1 \leq p \leq m$.

The first issue can be mitigated against by using a LEFT OUTER JOIN for annotation tables. However, solving the second problem requires a case differentiation in the SELECT clause using CASE ... WHEN ... THEN statements which quickly gets very complicated the more alternatives a query has.

A much simpler solution is to pad the SELECT clause of q_1 with NULL values and append the results for q_1 and q_2 using UNION as shown in Listing 2.

4.5 Corpus selection

Until now we always searched the entire database when looking for solutions to a query. This is fine for single-user systems but Annis was designed with many users in mind and the frontend only exposes those corpora which a particular user is allowed to use. We thus need a way to filter for (top-level) corpora which are referenced by the query (generated from the frontend) by their (unique) names.

If a query is to be performed against a document d then every document below d in the corpus hierarchy has to be searched as well. The node table is connected to the corpus table via the corpus_ref foreign key. To restrict the search to a particular document called *name* and all its children, we can use the following condition for each node table alias i used in the query:

```
nodei.corpus_ref IN (SELECT child.id FROM corpus AS child, corpus AS doc
                     WHERE child.pre BETWEEN doc.pre AND doc.post
                     AND doc.name = 'name')
```

---

[9] This case is arguably rare for node_annotation but it happens quite often for edge_annotation.

Listing 2: SQL query template for Annis queries with multiple alternatives using UNION.

```
SELECT DISTINCT
      node1.id, node2.id, ..., nodeN.id, NULL, ..., NULL
FROM                                       m − n times
      node AS node1 JOIN ...,
      node AS node2 JOIN ...,
      ...,
      node AS nodeN JOIN ...
WHERE
      conditions for q1

UNION SELECT DISTINCT
      node1.id, node2.id, ..., nodeM.id
FROM
      node AS node1 JOIN ...,
      node AS node2 JOIN ...,
      ...,
      node AS nodeM JOIN ...
WHERE
      conditions for q2
```

However, the frontend only exposes top-level corpora. During corpus import we have extended the `node` table with an attribute `toplevel_corpus` which links each node v to the top-level corpus of the document containing v. We can therefore implement the selection of corpora with a constraint on this attribute.

To enhance readability, we create a view of the `node` table containing only the nodes of the requested corpus *name* and refer to this view instead of the `node` table in the query. We need to wrap the creation of the view and the execution of the query inside a transaction so that multiple queries against the database can be run concurrently without manually managing view names:

```
BEGIN;
CREATE VIEW node_v AS SELECT * FROM node WHERE node.toplevel_corpus = 'name';
SELECT
      node1.id, ...
FROM
      node_v AS node1 ...
...
ROLLBACK;
```

4.6 Meta data filtering

For queries containing a meta data specification we only want to search documents below the requested root document that are properly annotated. We can piggy-back this condition on the view created in section 4.5. For example, the term `meta::lang:l1="de"` creates the following view:

```
CREATE VIEW node_v AS SELECT *
FROM node JOIN corpus_annotation AS corpus_annotation1
  ON (node.corpus_ref = corpus_annotation1.corpus_ref)
WHERE node.toplevel_corpus = 'name'
AND corpus_annotation1.namespace = 'lang'
AND corpus_annotation1.name = 'l1'
AND corpus_annotation1.value = 'de';
```

If the Annis query contains multiple meta data annotations, we have to join a separate corpus_annotation table alias to the node table for every listed annotation.

The SQL generation for an Annis query is now complete. In the next section we will describe how the query functions defined in section 3.7 are implemented.

4.7 Query functions

4.7.1 The $COUNT$ function

The $COUNT$ function is implemented by wrapping the original query as a subquery and counting the solutions using `count(*)` in the outer query.

4.7.2 The $ANNOTATE$ function

The $ANNOTATE$ function can be implemented by using the SQL query generated for an Annis query q as a subquery to generate the solutions S to q and then retrieve the overlapping tokens of the annotation graph fragment over S in the outer query. For this to work we need to modify the SELECT clause of the inner query to return the attributes needed to implement the overlapping operator:

```
SELECT DISTINCT
 node1.id AS id1,
  node1.text_ref AS text1, node1.left_token - left AS min1, node1.right_token + right AS max1,
 node2.id AS id2,
  node2.text_ref AS text2, node2.left_token - left AS min2, node2.right_token + right AS max2,
 ...,
 nodeN.id AS idN,
  nodeN.text_ref AS textN, nodeN.left_token - left AS minN, nodeN.right_token + right AS maxN
```

In the SQL fragment above, $left$ and $right$ specify how much context the annotation graph fragment returned by $ANNOTATE$ should contain. The outer query is depicted in Listing 3. Three features are worth mentioning:

1. The attributes node.id of the spans in a query solution are concatenated to create a key which groups all the spans of a retrieved annotation graph fragment (line 2).
2. We use OFFSET and LIMIT to retrieve only the annotation graph fragments for a subset of the query solutions to enable the pagination feature of the frontend described in section 3.8 (line 7). This requires that the result returned by the inner query is sorted which is achieved by the ORDER BY clause.
3. The result of the outer 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 24).

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 DDD*query* result described in [28]. 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 we have already seen 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.

4.7.3 The $MATRIX$ function

To implement the $MATRIX$ function, we need to modify the SELECT clause to retrieve the node annotations belonging to a span. If we simply retrieved the attributes of the node_annotation table for

Listing 3: SQL query used by the $ANNOTATE$ function to retrieve the annotation graph fragments over the solutions to a query q.

```
SELECT DISTINCT
  (matches.id1 || ',' || ... || ',' || matches.idN) AS key,
  facts.*
FROM
  (
    SQL query generated for q with modified SELECT clause
    ORDER BY id1, ..., idN OFFSET offset LIMIT limit
  ) AS matches,
  (
    node
      JOIN rank ON (rank.node_ref = node.id)
      JOIN component ON (rank.component_ref = component.id)
      JOIN node_annotation ON (node_annotation.node_ref = node.id)
      JOIN edge_annotation ON (edge_annotation.rank_ref = rank.pre)
  ) AS facts
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
```

each span, the result set would quickly grow very large. For example, if a query q contains n search terms and each span has m annotations, we would need m^n rows in the result set for one solution alone. Furthermore, the set semantics of the original SQL query, i.e. one row representing one solution to q, would be lost complicating the application code that has to parse the result set. A solution to this problem is to aggregate the annotations for one span into an array using the PostgreSQL-specific aggregate function `array_agg`:

```
SELECT
  node1.id AS id1,
  substr(text1.text, node1.left, node1.right - node1.left + 1) AS span1,
  array_agg(DISTINCT coalesce(node_annotation1.namespace || ':', '') ||
    node_annotation1.name || '=' || node_annotation1.value),
  ...
  nodeN.id AS idN,
  substr(textN.text, nodeN.left, nodeN.right - nodeN.left + 1) AS spanN,
  array_agg(DISTINCT coalesce(node_annotationN.namespace || ':', '') ||
    node_annotationN.name || '=' || node_annotationN.value)
...
GROUP BY id1, span1, ..., idN, spanN
```

We also retrieve the covered text for each span in accordance to the definition of the $MATRIX$ function. The `coalesce` function is used to ensure correct results in case the `namespace` attribute is set to `NULL`.

The result set of this query can be transformed into an annotation matrix using the algorithm described in definition 11.

5 Related work

As we have mentioned in section 3, the Annis query language is influenced by NiteQl and TIGERSearch. The NITE XML Toolkit uses Apache Xerces as its XML-processing backend [9, 3], but TIGERSearch is implemented as a custom application written in Java. In this section, we will briefly compare TIGERSearch to Annis. Because Annis queries are first translated to DDD*query*, which is based on XPath, we will also briefly discuss the research on evaluating XPath queries on relational database hosts.

5.1 TIGERSearch

The TIGERSearch query language was developed as part of the TIGER Treebank, a corpus of 40000 syntactically annotated sentences from German newspapers [8]. Similarly to Annis, it models sentences as two-dimensional trees: the syntax structure of a sentence is encoded by parent-child relationships of nodes and the word order is encoded by explicitly ordering the tree's leaves. Additionally, the TIGER data model allows for secondary edges between nodes if the syntax structure cannot be adequately captured by a tree.

A major difference to Annis is that each text span is represented by exactly one node. This is most obvious when multiple attributes of a span are queried at once. For example, the following Annis query looks for the possessive form of the German article *die*: `"der" & morph="Gen.Sg.Fem" & #1 _=_ #2`. Using TIGERSearch, this query can be shortened to: `[word="der" & morph="Gen.Sg.Fem"]`. TIGERSearch has no need for coverage operations because it works on unambiguously annotated corpora, whereas Annis supports corpora with conflicting annotations.

As with Annis, nodes are assembled into a graph template using dominance and precedence relations and constraints such as node arity. TIGERSearch supports negation for attribute values and node relations. Originally, it did not support universal quantification for negated values, but this was added as part of the Stockholm TreeAligner, a search tool for parallel treebanks based on TIGERSearch [22, 30].

An interesting feature of TIGERSearch is that a subset of the query language is also used as the corpus description language for the TIGER Treebank. This design strongly influences the implementation of the query processor. It is based on logical programming languages, specifically on the resolution of Horn clauses: given a graph g and a query q, TIGERSearch tests if it can find a set of nodes so that $g \cup q$ is contradiction-free [21].

Before a corpus can be searched it must be preprocessed. This generates the *index* – a proprietary, domain-specific column store. Each attribute value is stored in an attribute-specific list which is indexed by the node id. Furthermore, attribute values are dictionary-encoded to reduce space. The index also contains lists for other node properties, such as node arity and continuity and the node id of the left-most and right-most leaf to allow a quick implementation of left-most and right-most dominance as well as precedence. Finally, it contains the Gorn address of each node to quickly test for node dominance: a node s dominates another node t if the Gorn address of s is a real prefix of the Gorn address of t [13]. This index is conceptionally similar to the materialized `facts` table introduced in section 6.3.

The discussion of the index data structure shows that the implementation of the precedence operator is almost identical to Annis. The only difference is that TIGERSearch reduces precedence of non-terminal nodes to the left-most leaves whereas Annis takes both the left-most and the right-most leaves into account. The dominance operator is implemented differently in TIGERSearch and Annis, but both concepts are just as expressive. Gorn addressing, originally developed to encode a tree structure can be adapted to ODAGs in the same way as pre/post-order addressing was adapted for DDD*query*: A node which can be reached by more than one path from a root will have multiple Gorn addresses.

During the preprocessing phase of a corpus, TIGERSearch also generates statistics about the selectivity of each attribute value by building an inverted list, pointing from an attribute value to the containing graphs, for values below a configurable frequency.[10] Then, before evaluating a query, it first determines

---

[10] In the TIGER Treebank, a corpus normally consists of multiple graphs, each representing a sentence.

the most restrictive query term and the graphs that are a match for this term. The evaluation of the entire query is then limited to those graphs.

5.2 Evaluating XPath queries using relational databases

In recent years there has been a wealth of research regarding the efficient evaluation of XPath queries on SQL hosts. Two different approaches can be identified: schema-based systems which use information derived from a DTD or XML Schema definition to create a relational representation of the data stored in XML documents and schema-oblivious systems which strive to find a relational representation of XML documents independent of any particular schema [20]. The pre/post-order scheme proposed for DDD*query* [28] was originally developed as part of the XPath Accelerator [15] and is an example of a schema-oblivious system. This is a good match to the requirement of Annis to store conflicting annotation graphs from multiple annotation tools without a predefined tag set.

The key observation of the XPath Accelerator is that the pre- and post-order values of a node in a XML tree partition the pre/post-plane of the document in four non-overlapping regions which directly correspond to the major XPath axes ancestor, descendant, preceding and following. DDD*query* retains the ancestor and descendant axes but redefines the preceding and following axes to refer to the position of a span in a linear text and not its position in a tree and/or graph over tokens from that text.

A number of index structures and join algorithms have been proposed to efficiently evaluate XPath location steps along the ancestor and descendant axes, such as the use of Patricia keys to encode root-to-leaf paths [11] or the Structural Join (also called Containment Query) to quickly find ancestor-descendant relationships between two ordered lists of candidate nodes [31, 5, 10]. Unfortunately, most of these schemes require changes to the underlying database kernel and thus were not feasible as part of the Annis project.

Of particular interest is the Staircase Join algorithm [17]. An implementation[11] exists for PostgreSQL and many of its ideas can be expressed in purely relational terms [23, 16]. As it turns out, however, it achieves its remarkable performance by exploiting the semi-join semantics of XPath. To compute the result of a XPath location step starting from a set of context nodes C, not all nodes in C have to be evaluated. Those that contribute no new nodes because their target nodes are contained in the target nodes of another node in C can be pruned. The same is not true in a DDD*query* expression. Here we are not only interested in the target nodes of the last location step but potentially in any nodes that were found along the entire DDD*query* path expression. Nevertheless, some of the ideas in [16] can be adapted for Annis, namely the use of partitioned B-Trees to narrow down candidate nodes while computing joins.

Fortunately, the mapping from Annis to DDD*query* uses the descendant axis almost exclusively to define operators that refer to a span's position in a graph. Using a combined pre/post-order scheme, a step down the descendant axis can be computed using a bounded range lookup on a single value which is efficiently supported by B-Trees. The only operator that maps to the ancestor axis is the common-ancestor operator which benefits from an early-out strategy and is further sped up by the explicit partitioning of the graph into connected components.

The MonetDB/XQuery processor was also adapted to linguistic corpora based on multiple stand-off XML documents much like PAULA [7, 6]. It can query documents larger than 1GB interactively but a direct comparison with Annis is difficult because of differences in the underlying XML structure.

---

[11] The implementation is for PostgreSQL 7.4 which has been outdated for quite a while now. This reflects the ongoing effort by the developers to implement efficient XPath processors on off-the-shelf SQL databases. A loop-lifted variant of the Staircase Join is used in MonetDB/XQuery.

6 Evaluation and Optimization

The goal of this section is to analyze the performance of Annis 2 and improve it to a point where the system can be used interactively. Specifically, we want to achieve an evaluation time of less than two seconds for typical queries on a large corpus on current consumer hardware.

We used the 12 test queries listed in Table 23 on page 62 which were provided by users of the Annis system. The queries were evaluated against the TIGER corpus, a fairly large and deep corpus that includes edge annotations. Each query was run ten times with other queries randomly interspersed within, simulating a random workload on a single corpus. Since the performance of a query is strongly dependent on the contents of the PostgreSQL cache, we generated a new workload for each experiment as to minimize the possibility of a favorable query order skewing the results.

The evaluation times reported in this section are averaged over ten runs and were measured by the Annis client. They include the processing of the Annis query and generation of SQL code by the Annis compiler, the evaluation of the SQL query by PostgreSQL, and the transfer and processing of the database result set into an Annis data structure. In the case of $COUNT$ queries, where the result is a single integer, the last step is negligible. For the $MATRIX$ and $ANNOTATE$ queries in section 6.5 and section 6.6 it can generate significant overhead and may fail if the Java process has insufficient resources. Note that the rendering of the result in the web frontend is not included in the reported evaluation times.

More information about the experimental setup and the TIGER corpus is provided in appendix D.

6.1 Search boundaries for ranged operators

Before we measure the performance of Annis we should ensure that the generated SQL query contains as much information as can be derived from the original Annis query. Particularly, the linguistic operators listed in the last column of Table 7 require a ranged constraint on a table attribute which is only bounded in one direction. If possible, we should provide a second bound in order to minimize the number of tuples that need to be searched by PostgreSQL.

We have already seen one such transformation for the dominance operator in section 4.3.7:

$$\begin{aligned} j \text{ is a descendant of } i &\iff i_{\text{pre}} < j_{\text{pre}} \wedge i_{\text{post}} > j_{\text{post}} & \text{(8a)} \\ &\iff i_{\text{pre}} < j_{\text{pre}} < i_{\text{post}} & \text{(8b)} \\ &\iff i_{\text{pre}} < j_{\text{post}} < i_{\text{post}} \end{aligned}$$

In Equation 8a the search on the `pre` (or `post`) attribute is bounded in only one direction whereas Equation 8b provides both an upper and lower bound for the `pre` attribute.

The same transformation can be applied to the inclusion operator, however both `left` and `right` have to be bounded:

$$\begin{aligned} i \text{ includes } j &\iff i_{\text{left}} \leq j_{\text{left}} \wedge i_{\text{right}} \geq j_{\text{right}} \\ &\iff i_{\text{left}} < j_{\text{left}} < i_{\text{right}} \wedge i_{\text{left}} < j_{\text{right}} < i_{\text{right}} \end{aligned}$$

Query 6 of our test set uses the inclusion operator but no improvements are measurable when using the bounded implementation. However, the execution plan generated by PostgreSQL evaluates the inclusion predicates as a filter on the join that computes the parent operator in the query. This is also the top-most join and thus its input sets are already considerably restricted. If the query is simplified to `cat="VP" & tok & #1 _i_ #2` the effect of the optimization becomes apparent as shown in Figure 11.

Although `right` is unbounded in the left-overlap operator, PostgreSQL can efficiently perform the node join on the bounded `left` attribute.

Table 7: Table attributes required for the evaluation of Annis 2 language features. An attribute may be accessed using an equality or a ranged predicate depending on operator variant. The last column lists operators which evaluate at least one unbounded table attribute.

Language feature	**Equality access**	**Ranged access**	**Unbounded access**
Token search	node.span		
Text search	node.span	node.span[13]	
Annotation search	node_annotation.namespace, node_annotation.name, node_annotation.value	node_annotation.value[13]	
Edge annotation	edge_annotation.namespace, edge_annotation.name, edge_annotation.value	edge_annotation.value[13]	
Coverage	node.text_ref, node.left, node.right	node.left, node.right	`_i_ _ol_ _or_ _o_`
Precedence	node.text_ref, node.left_token, node.right_token	node.left_token, node.right_token	`.*`
Dominance, Pointing relations	rank.pre, rank.post, rank.parent, rank.level, component.type, component.name	rank.pre, rank.post, rank.level	`$*`
Root	rank.root		
Node arity	rank.parent		
Token arity	node.left_token, node.right_token		

The common ancestor operator contains an unbounded search of either the pre or post attribute in a correlated subquery. This is normally an indicator for bad performance but the subquery is guarded by a constraint on component.id and can be skipped for the majority of node joins. If the subquery has to be evaluated only the nodes in one component have to be checked.[12]

Finally, the indirect precedence operator and the general overlap operator cannot be bounded. When evaluating the term `#i .* #j`, for a given span i, any span following i satisfies the precedence constraint.[14] Similarly, when evaluating the term `#i _o_ #j`, a given span i only provides a lower boundary for the j_{right} and an upper boundary for j_{left} as Table 1 in section 3.3.1 shows.

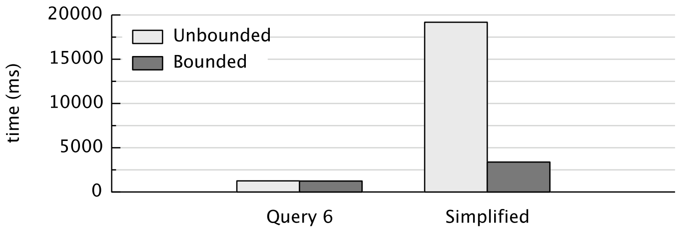


Figure 11: Effect of the inclusion optimization on query 6 and the simplified version `cat="VP" & tok & #1 _i_ #2`. In Tiger, 24962 nodes are annotated with `cat="VP"`.

---

[12] On average, there are about 11 nodes per component in the Tiger corpus.

[13] In some cases PostgreSQL can use an index for a regular expression predicate. See section 6.7 for details.

[14] The system allows to arbitrarily restrict the distance of spans that are considered for indirect precedence; however, in our test queries this did not provide a measurable speed-up, presumably because the database used other information contained in the query to sufficiently reduce the number of candidates for j.

6.2 Performance of the normalized corpus data model

We obtained a baseline for the evaluation of different optimization strategies by creating a B-Tree index for each attribute of the tables node, rank, component, node_annotation and edge_annotation and measured the performance of the *COUNT* query function. This initial test shows promising results: six of the test queries finish in less than two seconds. Queries that contain dominance operations however take significantly longer to complete and query 9 had to be aborted because it did not complete within 60 seconds.

The reason for this behavior becomes apparent if we analyze the join plan for query 9 which is shown in Figure 12. PostgreSQL has to join 16 tables to evaluate a query with only four search terms. The node table only contains the information necessary to evaluate a node or text search as well as coverage and precedence operations. For an annotation search the node_annotation table has to be joined. If the search term is referenced in a dominance or pointing relation operation both the rank and component tables have to be joined. Finally, if the dominance or pointing relation operation is qualified with an edge annotation, the edge_annotation table has to be joined as well.

As a result, to evaluate a single search term and the linguistic constraints that refer to it, PostgreSQL has to evaluate the predicates for each table alias separately and then join potentially large intermediary results.[15]

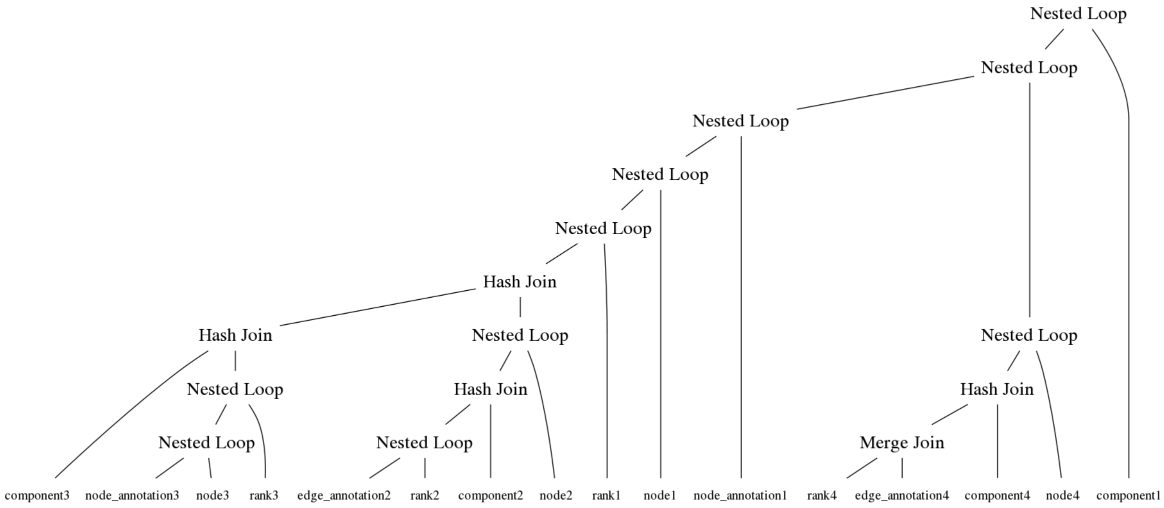


Figure 12: Join plan generated by PostgreSQL for query 9.

6.3 The materialized facts table

To reduce the number of joins we would like to access only one table alias for each search term of an Annis query. This can be achieved by joining the tables node, rank, component, node_annotation and edge_annotation and materializing the result as a facts table. To obtain a unique name for each attribute of facts we prefixed its name with the name of the original source table if it is ambiguous.

Again, we created an index for each attribute of the facts table and compared the performance of *COUNT* against the normalized source tables (Figure 13). The results are mixed: While the evaluation of queries containing many dominance operations such as query 9 is accelerated considerably, queries that contain regular expressions or many operations requiring only the node table such as query 1 are faster when performed on the normalized source tables.

---

[15] The problem is aggravated by the use of a genetic query optimization algorithm by PostgreSQL for queries with more than 12 tables in the FROM clause which can produce non-deterministic results.

The facts table negatively affects query performance in two ways: First, the size on disk of the TIGER corpus is more than doubled as Table 8 shows. Secondly, each node is represented multiple times in the facts table which increases the search space when computing search terms and linguistic constraints.[16]

Nevertheless, the materialized facts table is necessary to provide optimized indexes which we discuss in the next section. The influence of regular expressions and their evaluation by PostgreSQL is discussed in section 6.7.

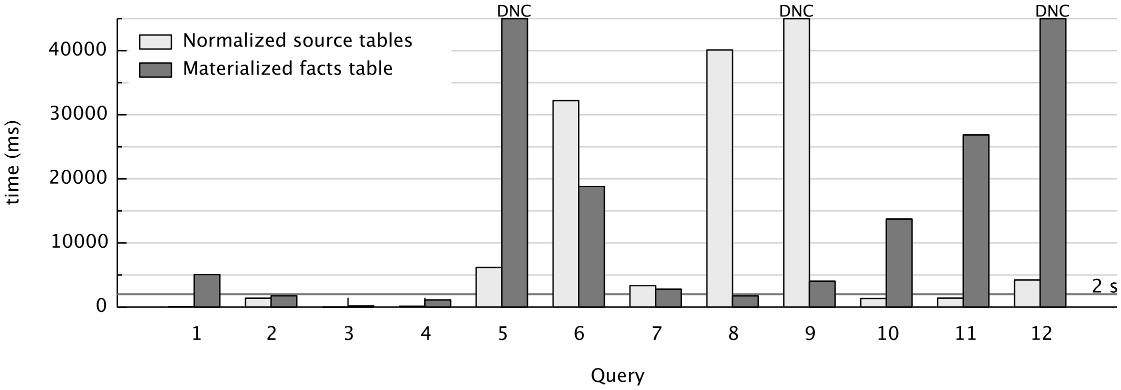


Figure 13: Performance of $COUNT$ on the normalized source tables and the materialized facts table. Queries 5, 9 and 12 did not complete within 60 seconds.

Table 8: Size of the TIGER corpus on disk (in MB).

	Total	**Tables**	**Indexes**
Source tables	1536	428	1099
facts table	3400	756	2638

6.4 Combined node lookup and node join

Let us review the execution plan generated by PostgreSQL for query 5 which is depicted in Figure 14. Although the query contains four annotations searches, only one of them is matched by consulting an index over the appropriate attributes. The others are computed by first finding all nodes that satisfy a linguistic constraint – using the index over pre for the dominance operation and the indexes over text_ref, left and right_token for coverage and precedence – and then filtering for those nodes that match the second search term in the linguistic constraint. PostgreSQL effectively discards much of the information contained in the query when computing intermediate result sets.

Disabling nested loop joins turns the situation upside down: As Figure 15 shows indexes are consulted to match search terms and linguistic constraints are evaluated in joins. PostgreSQL still does not use all the information provided in the query when scanning indexes and additionally a number of costly hashing operations[17] are introduced.

Consider a combined index over span, text_ref and left: It can be used to compute the result of a text search and will return the tuples sorted in such a way that a merge join can be used to evaluate an inclusion, left-align or left-overlap operation in the same query. Ideally, we would like to construct such an index for any combination of search term and linguistic constraint to enable PostgreSQL to use as much information as possible when scanning indexes.

---

[16] Generally the number of tuples in facts for each node is $n \times e \times p$ where n is the number of node annotations, e the number of edge annotations and p the number of parents of the node.

[17] Or sorting operations in other queries.

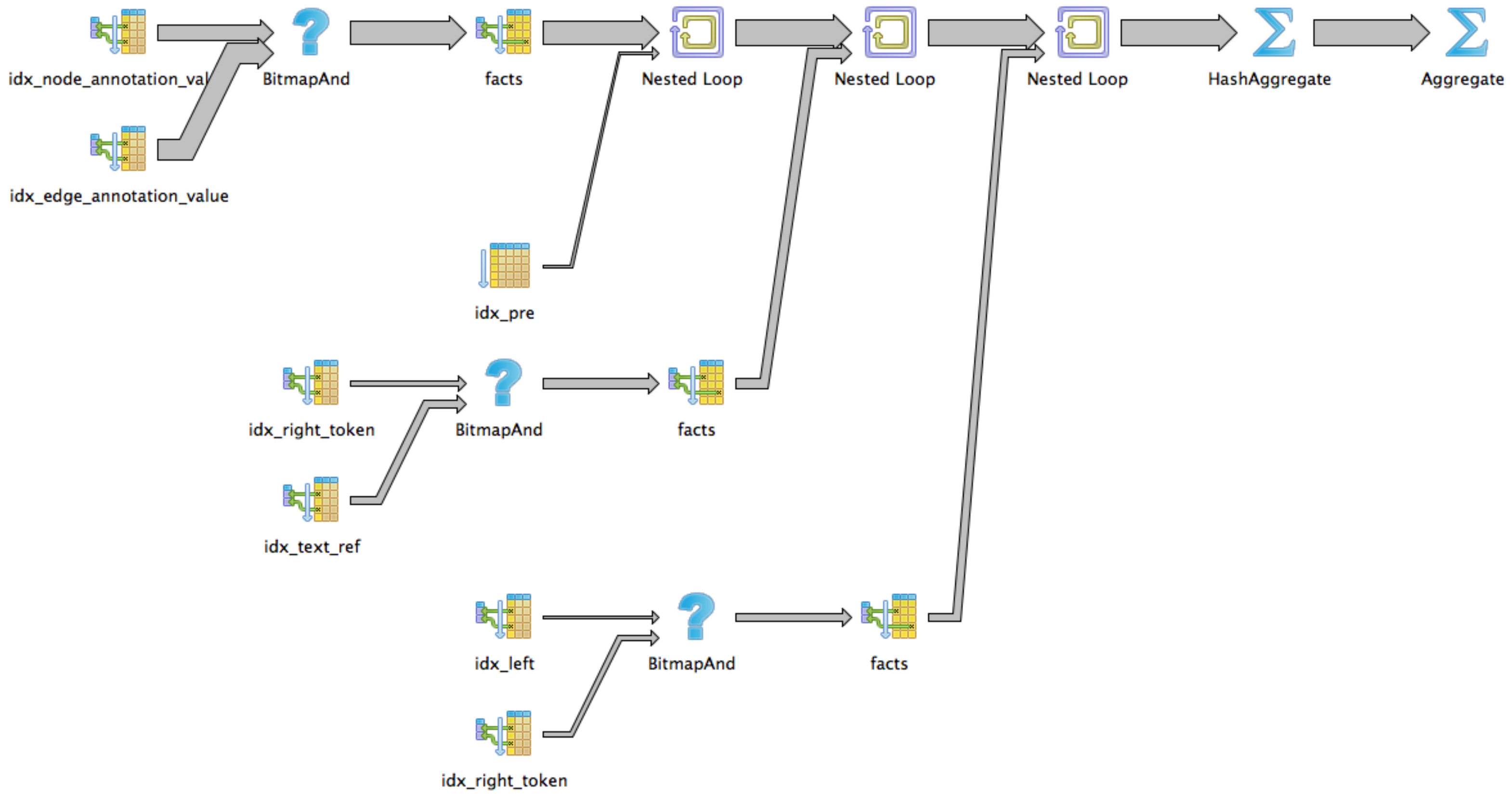


Figure 14: Execution plan generated by PostgreSQL for query 5 with nested loops joins enabled.

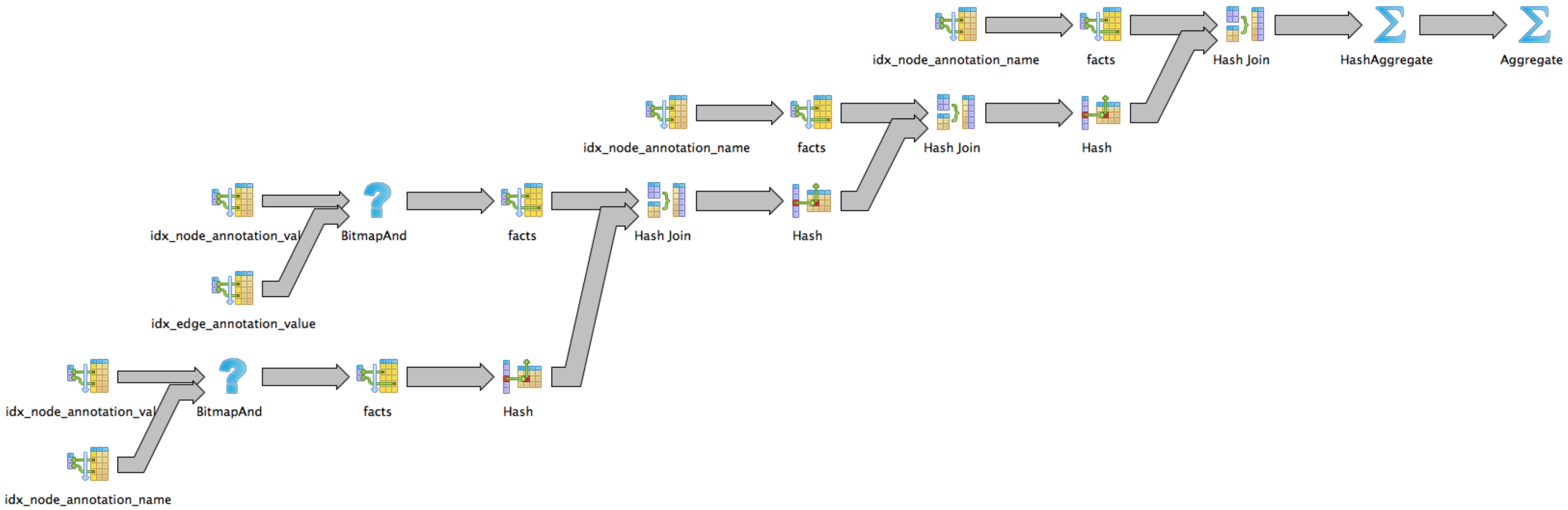


Figure 15: Execution plan generated by PostgreSQL for query 5 with nested loops joins disabled.

6.4.1 Indexed attributes for search terms and linguistic constraints

Search terms in Annis are translated into value constraints on table attributes; linguistic constraints generally into joins over columns specifying the position of the node in the graph or linear text. Both node orders are partitioned – the entire corpus into texts and the annotation graphs over these texts into components. Indexes therefore have to start with (a combination of) table attributes for a node lookup and end with (a combination of) attributes matching conditions of a linguistic node join. Additionally, index fragments implementing node joins should start with text_ref or component_id.[18] Table 9 lists the attributes indexed for search terms and linguistic constraints.

Table 9: Indexed attributes for search terms and linguistic constraints.

Language feature	**Indexed attributes**
Text and token search	span
Annotation search	node_annotation_name / node_annotation_value *or* node_annotation_name / node_annotation_name, node_annotation_value
Coverage	text_ref, left / text_ref, right / text_ref, right, left
Precedence	text_ref, right_token / text_ref, left_token − 1
Dominance and pointing relations	pre / parent / component_id

An index prefixed with node_annotation_name and node_annotation_value can only match annotation searches efficiently if the annotation value is provided in the query. For annotation searches such as e.g. cat a second index over node_annotation_name is required. Alternatively, we could construct one index prefixed with node_annotation_name and a second prefixed with node_annotation_value to reduce the size of the latter. This is motivated by the distribution of node annotation values; the vast majority uniquely identify the corresponding annotation name.

Prefixing the index sets with node_annotation_namespace will not improve performance. First of all, in the TIGER corpus each annotation is prefixed with *tiger*, so this attribute does not influence the selectivity of the query at all for this particular corpus. Secondly, if a query contains an annotation search without a namespace these indexes cannot be used anyway. We thus need a second set of indexes without node_annotation_namespace which more than doubles index construction time during corpus import and the size used by the indexes on disk. Thirdly, even if every annotation search in a query specifies a namespace, PostgreSQL will actually scan both versions of an index – in the same execution plan! – a particularly unenlightened decision by the query planner which reduces the buffer cache size substantially.[19]

Finally, we should note that edge annotations conceptionally are search terms as well; instead of nodes they select edges. We can construct indexes prefixed with edge annotation attributes and ending with pre or parent; these can be used by PostgreSQL in conjunction with the node annotation indexes to narrow down candidate nodes for annotated edge operations in the query. Figure 15 contains an example of such a bitmapped index access.

Precedence operations can use an index over text_ref and right_token. An index over text_ref and left_token can only be used for the indirect precedence operator because PostgreSQL is unable to evaluate the expression left_token − 1 on it. Fortunately, PostgreSQL supports indexes over expressions containing

---

[18] PostgreSQL's implementation of multicolumn B-Tree indexes is particularly efficient when the leading (left-most) n columns are used in an equality predicate. If column $n + 1$ is used in an inequality or ranged predicate it is also used to limit the portion of the index that has to be scanned. If an index is defined on more than $n + 1$ columns the index is used to check any predicates that refer to these columns, however the part of the index that has to be scanned is not reduced by them.

[19] For query 8, PostgreSQL chose the index over pre prefixed with node_annotation_namespace to match `tiger:pos="PRELS"` and the index without the namespace to match `tiger:pos="NE"`.

Table 10: Subset definitions for partial indexes.

Search term	**WHERE clause**
Text and token search	span IS NOT NULL
Annotation search	`node_annotation_name = 'name'`
Dominance operations	`edge_type = 'd'` / `edge_type = 'd' and edge_annotation_name = 'name'`

table attributes allowing us to construct an index over text_ref and left_token − 1. To make this index usable for indirect precedence we have to change the implementation of the operator as follows:

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

The index on component_id does not need to be prefixed with table attributes for search terms because it is only used to look up the (anonymous) common ancestor in said operation.

6.4.2 Partial indexes

PostgreSQL supports partial indexes, i.e. indexes over a subset of tuples that satisfy the conditions of a WHERE clause [27]. Partial indexes are useful on attributes that are used in a value constraint in a query and where the distribution of values is known in advance, such as attributes discriminating the tuples of a table into two or more classes [26, 4].

The SQL queries generated by Annis present many opportunities to construct partial indexes which are summarized in Table 10. Obviously, the edge_type attribute partitions the facts table into distinct regions of which only one is of interest when evaluating dominance or pointing relation operations. This is even useful for the TIGER corpus which does not contain pointing relations because it eliminates coverage edges and unconnected nodes.[20]

The text and token search are only interested in tokens, i.e. nodes where span is not NULL. We could configure the index to store NULLs last and reduce the section of the index that has to be scanned but in that case a large part of the index will *never* be scanned and simply waste disk space.[21]

Finally, since there is typically only a limited set of annotation names in a corpus, we can create indexes dedicated to a particular annotation name. If an index contains multiple annotation namespaces we could further partition these into dedicated indexes for fully qualified names.

6.4.3 Evaluation of different indexing strategies

We created the following three sets of indexes and measured their performance on a random workload:

1. A *value-only* indexing strategy with indexes prefixed separately with node_annotation_name and node_annotation_value containing 33 indexes.
2. A *qualified* indexing strategy where the index over node_annotation_value was additionally prefixed with node_annotation_name.
3. A *partial* indexing strategy as described above containing 80 indexes.

Table 11 contains the average runtime of each query on a random workload for the different strategies. Of the three approaches, the partial strategy is clearly the superior one. With the exception of queries dominated by regular expression searches, it out-performs the evaluation time on the source tables or is

---

[20] This eliminates about 7% of the tuples in facts.

[21] About a third of the nodes in TIGER are non-tokens.

only a little slower. Eight of the queries complete in well under two seconds and query 6 is not much slower with 2.6 seconds on average. The value-only and the qualified indexing strategy appear to affect query performance equally. We suspect that the generally small difference in the runtimes for individual queries is a consequence of the random nature of the experiment.

These findings are supported by the disk utilization during the experiment which is shown in Table 12. The partial indexes require about the same space as the other indexes on disk, however the amount of data read during the experiment is cut in half. It is worth pointing out that compared to the normalized source tables, PostgreSQL had to read four times as much data during the experiment on the partial indexes. This validates the approach of providing dedicated indexes for combined node lookup and join: Although the number of indexes results in a higher frequency of buffer cache misses, the time spent in joins is reduced considerably.

Finally, Figure 16 compares the average runtime of each query with the best time for five sequential runs for the partial indexing strategy. Slow queries with regular expressions are somewhat faster but they still perform poorly on the facts table.

Table 11: Average query evaluation times depending on indexing strategy (in ms). The best time is listed in bold for each query. Missing values indicate queries where at least three runs did not complete in 60 seconds.

Query	Source tables	facts table	Value-only	Qualified	Partial
1	**84**	5061	755	904	315
2	1381	1749	1473	1599	**142**
3	**28**	181	134	105	37
4	**128**	1103	388	511	163
5	6183		12477	10203	**1197**
6	32209	18814	9108	10592	**2624**
7	3332	2783	1337	1204	**361**
8	40119	1735	537	1566	**175**
9		4030	2873	4026	**1511**
10	**1344**	13728	7533	5620	3494
11	**1399**	26850	14633	18352	3993
12	**4218**		38083	39670	9762

Table 12: Space requirements (in MB) and indexing times (in minutes) for different indexing strategies. Also shown is the amount of data read from and written to the disk during the experiment.

Strategy	Indexing time	Disk space	MB read	MB written
Source tables	1:53	1099	1558	20
facts table	7:27	2638	8915	341
Value-only	22:23	4770	11962	616
Qualified	15:51	4450	12601	660
Partial	11:35	4593	6413	45

6.5 The $MATRIX$ query function

Whereas $COUNT$ returns a single number, the $MATRIX$ function returns one row for each solution to a query. A little variation in the row width is introduced by the number of nodes in the query and varying sizes of text spans. As expected, the evaluation time of $MATRIX$ grows linearly with the number of solutions (Figure 17).

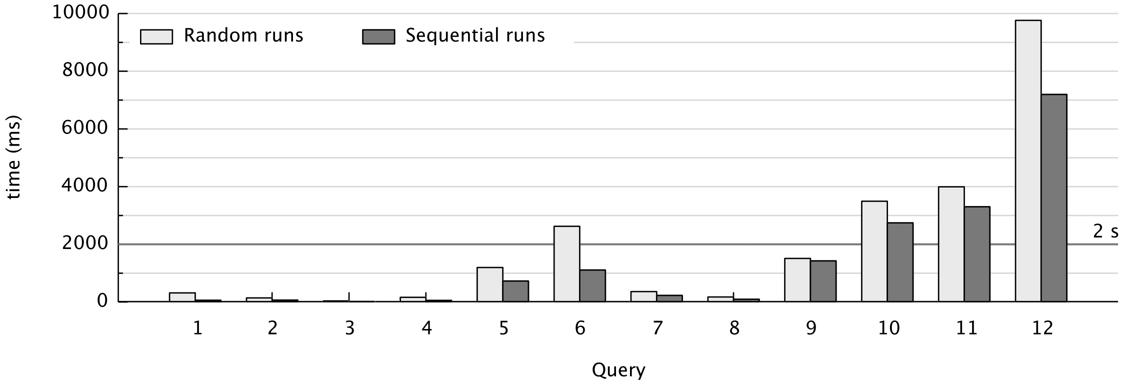


Figure 16: Comparison of the average runtime for each query in a random workload vs. the best runtime in five sequential runs using the partial indexing strategy.

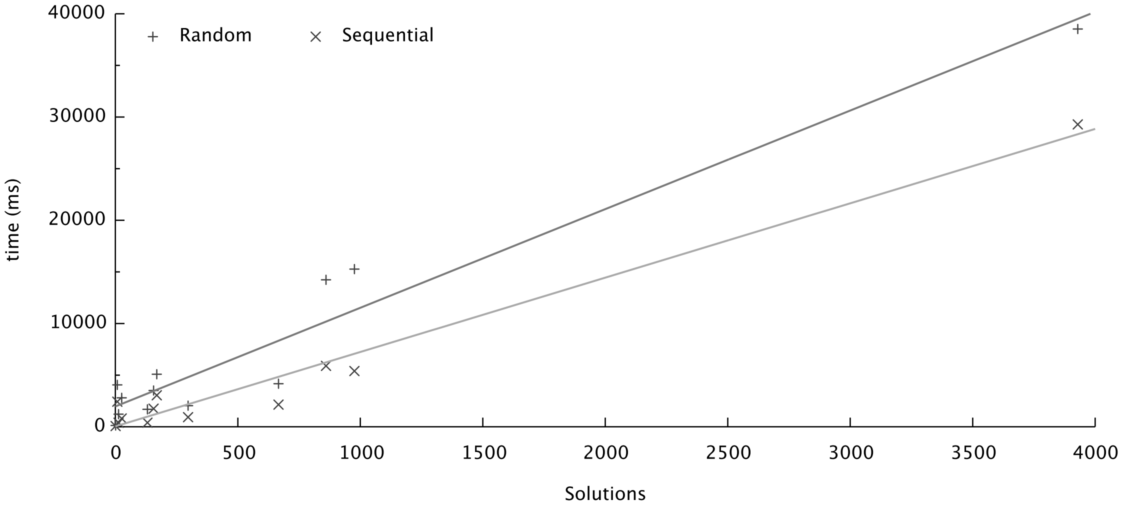


Figure 17: Evaluation time of the $MATRIX$ query function depending on the number of solutions to a query.

6.6 The $ANNOTATE$ query function

For each solution to a query the $ANNOTATE$ function returns multiple rows depending on the initial number of tokens covered and the requested context. Accordingly, the result set for an entire query can grow quite large as Table 13 shows for query 7; we observed multiple megabytes being transferred over the network for typical queries. The nearly linear growth of the row count is mirrored by the time required to evaluate the $ANNOTATE$ function which is depicted in Figure 18. For large values of *context* and *limit* the performance of the Annis service is dominated by the Java client and not the database. PostgreSQL evaluated the query `annotate tok` with $limit = 1000$ and $context = 100$ in about 30 seconds, while the Annis client did not complete within five minutes.

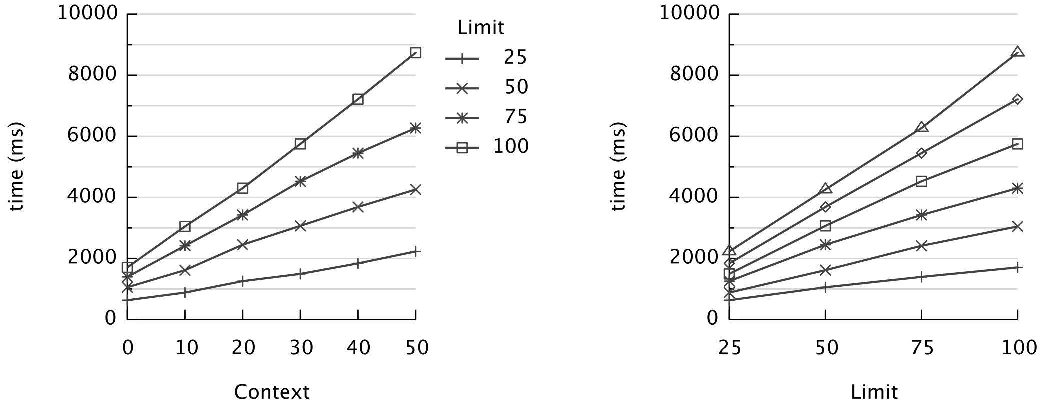


Figure 18: Influence of *limit* and *context* on $ANNOTATE$.

This behavior is acceptable because the information returned by $ANNOTATE$ is presented to the user at once. Thus, a large value for *limit* is not a sensible use case. For small values, $ANNOTATE$ does not take much longer than counting all solutions as Figure 19 shows. The performance of slow queries is actually improved a little which can be seen in Table 14 for query 12. Additionally, we observed little variation between the average runtime on a random workload and the best of five sequential runs, a finding that was accompanied by virtually no disk utilization during the entire $ANNOTATE$ experiment.

Table 13: Rows in the result set of the $ANNOTATE$ function for query 7 depending on the number of annotated solutions and the requested context.

Solutions	**Context**					
	0	**10**	**20**	**30**	**40**	**50**
25	2288	5517	8746	11925	14960	18035
50	6420	12921	19212	25354	31279	37249
75	9318	19056	28567	37852	46747	55635
100	11965	24995	37734	50179	62018	73798

Table 14: Values for *limit* for which $ANNOTATE$ is faster than a full count of query 12 (9762 ms).

Limit	**Context**			
	0	**10**	**20**	**30**
25	8692	8917	9254	9758
50	8748	9461		
75	8790	9663		
100	8915			

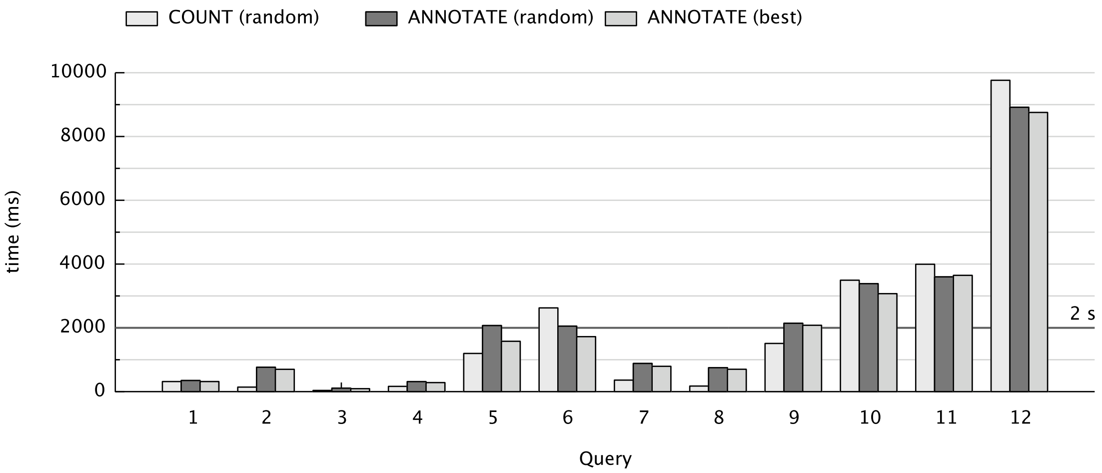


Figure 19: Runtime of $COUNT$ vs. $ANNOTATE$ with $limit = 25$, $context = 10$ on a random workload (middle) and the best runtime in five sequential runs (right).

6.7 Rewriting queries with anchored regular expression searches

PostgreSQL can use an index for a regular expression value constraint if the regular expression is anchored at the beginning and starts with a single character. Of the regular expressions encountered among the test queries only the annotation search `pos=/N.*/` satisfies that condition.[22]

This explains the slow performance of query 12. To PostgreSQL it provides the same information as the query below which is a complicated way of asking for tokens annotated with `pos="NN"`.

```
lemma & tok & pos="NN" & #1 _=_ #2 & 2 _=_ #3
```

Eliminating unanchored regular expressions can provide a substantial performance improvement. For example, in query 11 the text search `/[19][09][0-9][0-9]/` is looking for dates between 1900 and 2099. The query as stated in Table 23 results in an execution plan in which a slow scan on the `facts` table is filtered for matching spans. This outer scan drives a nested loop to match `pos=/N.*/` annotations (Figure 20).

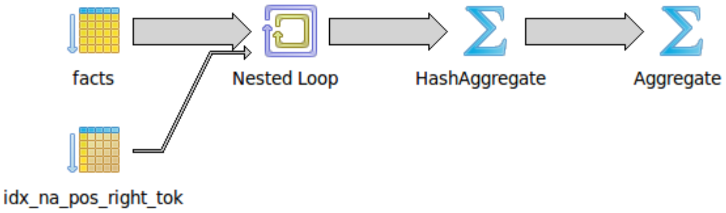


Figure 20: Execution plan for the unanchored regular expression search `/[19][09][0-9][0-9]/`.

We can rewrite the query using `OR` and enable PostgreSQL to scan an index for spans starting with `19` or `20` respectively (Figure 21):

```
pos=/N.*/ & ( /19[0-9][0-9]/ & #1 . #2 | /20[0-9][0-9]/ & #1 . #3 )
```

The second version performs much better as Figure 22 clearly shows. On a random workload it is 2.7 times faster and in sequential runs it completes in less than 150 milliseconds compared to more than three seconds for the unanchored version.

---

[22] Annis anchors regular expressions implicitly.

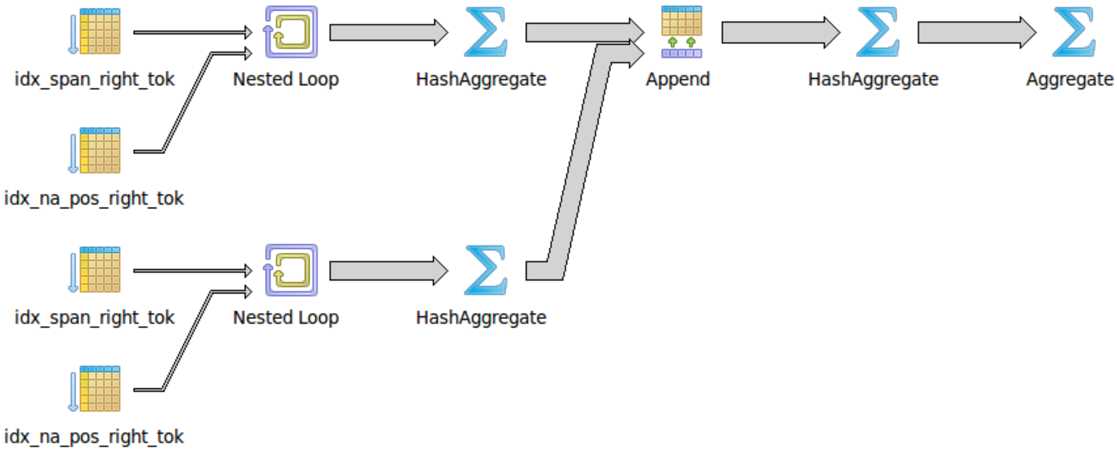


Figure 21: Execution plan for the anchored regular expression searches `/19[0-9][0-9]/` and `/20[0-9][0-9]/`.

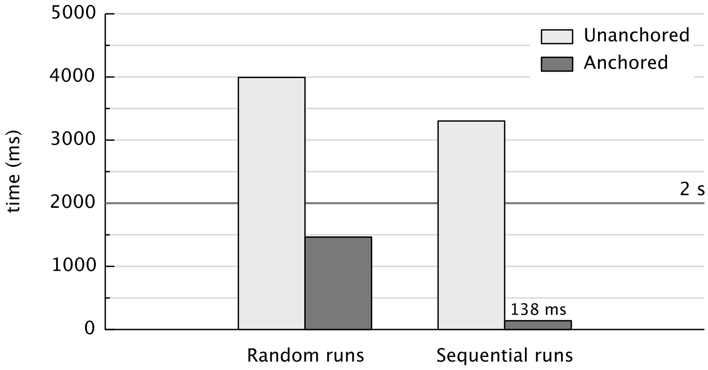


Figure 22: Performance of unanchored vs. anchored regular expressions.

6.8 Influence of document size

In a final experiment we doubled the size of the TIGER corpus which resulted in a `facts` table containing a little more than 8 million tuples and measured the performance of *COUNT*. The results are somewhat discouraging. As Figure 23 shows only query 3 completed in less than two seconds on average and queries containing many search terms with a low selectivity such as `node`, `cat` or unanchored regular expression searches did not complete within 60 seconds.

We suspect that this slowdown is mainly caused by PostgreSQL not having enough resources to handle a corpus of that size. This conclusion is supported by the amount of data read from disk during the experiment: more than 40 GB.

Queries containing many search terms with a low selectivity did not benefit much from the buffer cache. For example, query 5 contains two `cat` searches and required 42 and 35 seconds respectively to complete two consecutive runs. This is a stark contrast to the performance of query 4 with its highly selective `"desto"` and `morph="Comp"` search terms. While the first run required 21 seconds to complete the second iteration finished in 165 milliseconds – a speedup by two orders of magnitude. As we can see in Figure 24, the optimal performance of queries that do not contain search terms with a low selectivity is roughly linear in the size of the corpus.

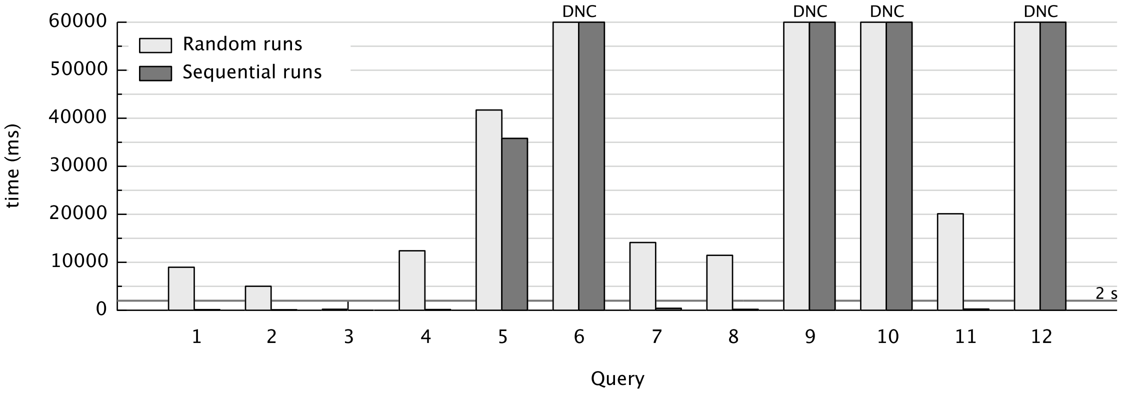


Figure 23: Comparison of the average runtime for each query in a random workload vs. the best runtime in five sequential runs on the 1 GB Tiger instance. Query 11 was substituted with the anchored version.

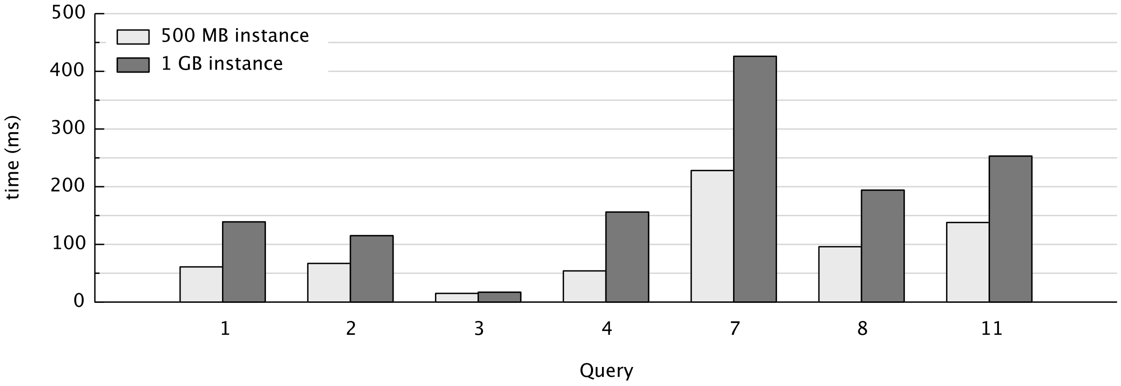


Figure 24: Best runtime in five sequential runs on the 500 MB and 1 GB Tiger instances.

7 Conclusions and Outlook

With the exception of queries dominated by unanchored regular expressions we achieved our goal set out at the beginning of section 6. The interactive use case of the *ANNOTATE* function, where the user is interested in retrieving *some* results *fast*, is well-supported since the evaluation of these queries is not particularly sensitive to the contents of the buffer cache and only marginally slower than counting all solutions. Annis can quickly present the first 25 results and then pre-fetch the next result set while the *COUNT* query completes in the background.

We can identify four techniques which improved query performance:

- Partitioning the annotation graph into connected components eliminates duplicate subgraphs contained in the rank table which are originally caused by nodes with multiple parents.
- The elimination of joins by materializing a facts table. In itself this causes many queries to perform slower but it enables us to construct indexes which use information from all source tables. It is worth pointing out that the XPath processors mentioned in section 5.2 also use a single (and much narrower) facts table.
- The combination of node lookup and node join in one index scan. Although the large number of combined indexes reduces the hit rate of the buffer cache, less time is spent in joins because the number of candidate tuples is reduced.
- Optimizing the cache hit rate by providing indexes partitioned by annotation name. This reduces the amount of data that has to be read from disk during the experiment by almost 50%.

With regards to DDD*query* we found that it is not particularly suited to evaluate Annis queries for two reasons:

- Edges are only weakly supported in DDD*query*. For example, alignment links between text spans are modeled as alignment nodes in [29] to which annotations can be attached. As a consequence the number of parents of a node increases on average. This is not necessarily a bad choice. In the original DDD*query* data model it would cause an explosion of the rank table but as we have shown it is possible to minimize the number of rank tuples that are attached to the same node.
- More importantly, the basic language feature of DDD*query* – a path definition – is not expressive enough to construct linguistic queries concisely. Annis queries are essentially subgraph templates which can contain cycles if one ignores the type of the linguistic constraint. Expressing these in terms of paths is cumbersome. We therefore chose to forego the implementation of advanced DDD*query* features such as regular path expressions and opted to simply list the nodes and edges of the subgraph as done in AQL2.

For these reasons and because the intermediate translation of an AQL2 query to DDD*query* causes a significant implementation overhead, future releases will skip this step and directly translate from AQL2 to SQL.

The work on Annis is ongoing and we plan to add more features in the future. Concerning the Annis service back-end we would like to mention the following ideas:

- Support for parallel corpora and alignment of text spans. This can be achieved by adding a new alignment edge type and providing operators which query these edges. A prototype implementing alignment links between nodes from separate texts using pointing relations is already working.
- We plan to generalize the two text orders contained in the model – characters for coverage and tokens for precedence – and extend the precedence operator with a precedence level much like edge operations can be qualified with a name. This would enable us to model subtokens such as syllables. Additionally, it would allow us to specify the context level for the *ANNOTATE* query function; instead of a context of n tokens we could retrieve the entire sentence or paragraph containing a match.

- Annis 2 only supports strings as annotation values. For corpus annotations specifically, we would like to support more data types in order to construct queries such as `meta::speaker-age < 20`. This feature can be added easily using a SQL `CAST` expression.
- We want to add negation to the query language. This addition would be two-fold: Search terms can be negated using the != operator. The negation of linguistic constraints, however, is much more complex and requires the introduction of an (implicit) all quantifier.
- Queries that only require data contained in the `node` table could benefit significantly if they were evaluated on this table alone and not the materialized `facts` table.
- Finally, we would like to add a full-text search to the query language as to easier search for phrases without resorting to queries containing many precedence operations.

A Annis 2 Query Language Grammar

A grammar for the Annis 2 query language is reproduced below. Note that the language definition imposes further constraints on valid queries as explained in definition 7.

⟨*query*⟩ ::= ⟨*expression*⟩

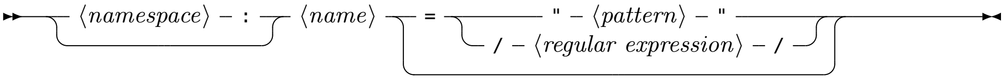
⟨*expression*⟩ ::=

⟨*search term*⟩ ::=

⟨*node search*⟩ ::= `node`

⟨*annotation search*⟩ ::= ⟨*annotation*⟩

⟨*annotation*⟩ ::=



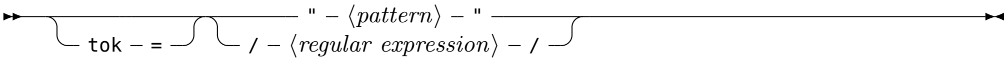
⟨*namespace*⟩ ::= A string containing alphabetic characters, digits and the symbols '_', '-' and '.', starting with a character, '_' or '-'.

⟨*name*⟩ ::= See definition for ⟨*namespace*⟩ above.

⟨*pattern*⟩ ::= A string containing any character except '"'.

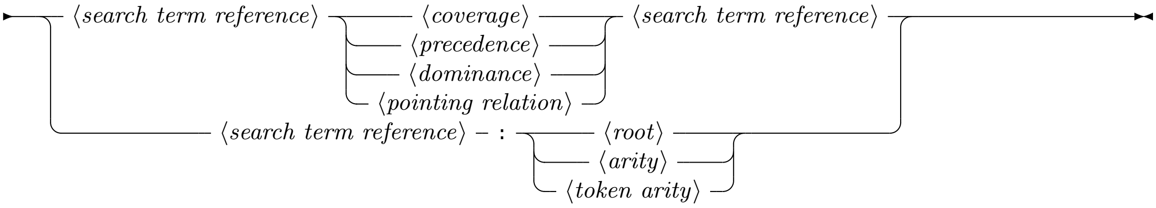
⟨*regular expression*⟩ ::= A string containing any character except '/'.

⟨*text search*⟩ ::=

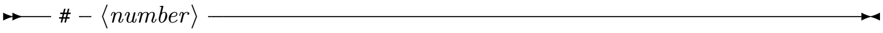


⟨*token search*⟩ ::= `tok`

⟨*linguistic constraint*⟩ ::=

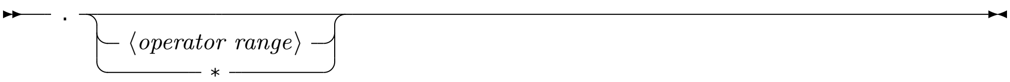


⟨*search term reference*⟩ ::=



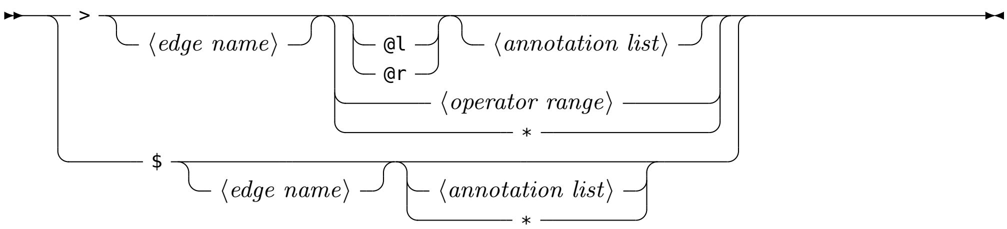
⟨*coverage*⟩ ::=

⟨*precedence*⟩ ::=



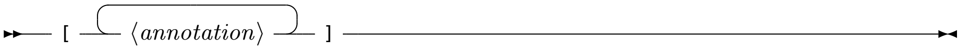
⟨operator range⟩ ::=

⟨dominance⟩ ::=

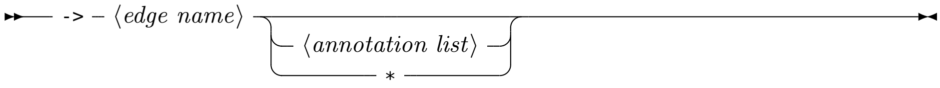


⟨edge name⟩ ::= See definition for *⟨namespace⟩* above.

⟨annotation list⟩ ::=

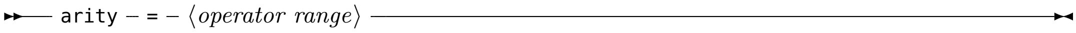


⟨pointing relation⟩ ::=

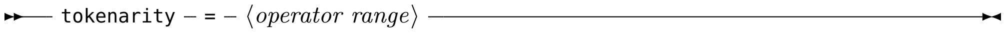


⟨root⟩ ::= `root`

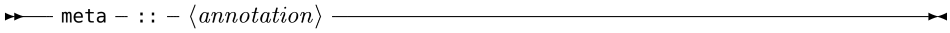
⟨arity⟩ ::=



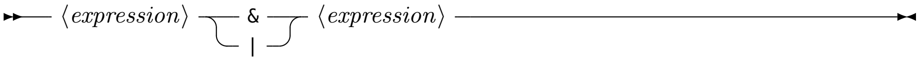
⟨token arity⟩ ::=



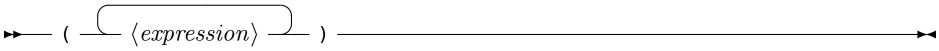
⟨meta constraint⟩ ::=



⟨boolean expression⟩ ::=



⟨expression group⟩ ::=



B Internal DDD*query* implementation

For historical reasons Annis 2 first transforms an AQL query into an intermediate DDD*query* before generating the final SQL output. The internal DDD*query* implementation used by Annis is incomplete; only a subset of the features that is required to implement Annis is supported.

Unfortunately the DDD*query* corpus model is a poor match for some Annis features and we had to extend DDD*query* considerably in order to answer queries efficiently. In the end, the DDD*query* language used internally by Annis 2 transformed into a close resemblance of AQL2 proper, albeit with a different syntax.

A complete DDD*query* grammar can be found in [29].

B.1 Supported DDD*query* features and custom extensions

Annis partially implements most of the DDD*query* feature set except for regular path expressions. Instead, a DDD*query* can contain multiple paths that are grouped with logical AND and OR as described in section 3.4.

The notion of a node type which DDD*query* retains from its XPath roots is meaningless within the Annis corpus model. Consequently, most features that make use of the node type, particularly different DDD*query* node tests, are missing in the internal DDD*query* implementation.

All constituents of a DDD*query* step are supported, including (nested) node set predicates, functions and binding of node sets to variables and names.

The node test element is used in its generic form for each search term to select any node. The attribute note test is used to filter a node set for annotation searches. Similarly, the node set selected by element is filtered by text value or the isToken function for text and token searches.

Annis adds a variable node test in the form of `$ni/axis::$nj` that is used to connect the two previously bound node sets `$ni` and `$nj` by any axis.[23]

The following DDD*query* axis are implemented as defined in the DDD*query* specification: attribute, following, immediately-following, containing, overlapping-following and overlapping-preceding.

The matching-element axis is redefined as selecting any node that covers the same text as a node from the context node set.

Additionally, the child and descendant axis can optionally be qualified with an edge type and edge name in the form `child[type]` or `child[type, name]`. Similarly, the sibling axis can be qualified with an edge name in the form `sibling[name]` (the sibling axis always selects dominance edges).

The descendant axis can further be qualified with an expected path length in the form `descendant(n)` or `descendant(n,m)`. The child axis can be qualified with a list of edge annotations in the form `child(namespace:name="value", ...)`.

Finally, Annis implements the following custom axis: overlapping, left-align, right-align and common-ancestor. Their usage is explained in Table 16.

Annis makes use of the following custom DDD*query* functions which are explained in Table 15 and Table 17: isToken, isRoot, arity and tokenArity.

---

[23] In the original DDD*query* specification the child axis is always implied in a step `$ni/$nj`.

B.2 Mapping from AQL2 to DDD*query*

A DDD*query* is build from an AQL2 query by substituting DDD*query* path expressions for search terms and linguistic constraints in the original Annis query. The logical structure of the query is retained.

Table 15 lists the DDD*query* expressions substituted for Annis search terms. The DDD*query* variable `$ni` denotes the ith search term in the original Annis query. The node set returned by the DDD*query* node test is bound to `$ni` in order to refer to it later in DDD*query* substitutions for linguistic expressions.

Table 15: DDD*query* mappings for Annis search terms.

	Annis search term	**DDDquery expression**
Node search	`node`	`element()#(ni)$ni`
Annotation search	`namespace:name="value"`	`element()#(ni)[@namespace:name="value"]$ni`
Text search	`"Mary"`	`element()#(ni)[. = "Mary"]$ni`
Token search	`tok`	`element()#(ni)[isToken()]$ni`

A regular expression `/Mary/` is translated as a DDD*query* regular expression `r"Mary"` in a text or annotation search.

Binary linguistic expressions `#i operator #j` are mapped by connecting the node sets referred to by the DDD*query* variables `$ni` and `$nj` with a operator-specific DDD*query* axis step.

All substitutions follow the template `#i operator #j` → `$ni/axis::$nj`. Table 16 lists the operator-specific DDD*query* axis substituted for each linguistic operator.

Table 16: DDD*query* axis mappings for binary Annis linguistic expression.

Operator	**Name**	**DDDquery axis**
	Coverage	
`#i _=_ #j`	Exact Cover	`matching-element`
`#i _i_ #j`	Inclusion	`containing`
`#i _l_ #j`	Left Align	`left-align`
`#i _r_ #j`	Right Align	`right-align`
`#i _ol_ #j`	Left Overlap	`overlapping-following`
`#i _or_ #j`	Right Overlap	`overlapping-preceding`
`#i _o_ #j`	Overlap	`overlapping`
	Precedence	
`#i . #j`	Direct precedence	`immediately-following`
`#i .* #j`	Indirect precedence	`following`
`#i .n,m #j`	Ranged precedence	`following(n,m)`
	Dominance	
`#i >name #j`	Direct dominance	`child[d, name]`
`#i >name * #j`	Indirect dominance	`descendant[d, name]`
`#i >name n,m #j`	Ranged dominance	`descendant[d, name](n,m)`
`#i >@l #j`	Left dominance	`child[l]`
`#i >@r #j`	Right dominance	`child[r]`
`#i $name #j`	Sibling	`sibling[name]`
`#i $name * #j`	Common ancestor	`common-ancestor[name]`
	Pointing relations	
`#i ->name #j`	Direct link	`child[p, name]`
`#i ->name * #j`	Indirect link	`descendant[p, name]`

Unary linguistic operators are implemented using the custom DDD*query* functions listed in Table 17 which are attached to the node set referred to by the DDD*query* variable `$ni`:

Table 17: DDD*query* mappings for unary Annis linguistic expressions.

	Annis term	**DDDquery expression**
Root node	`#i:root`	`$ni[isRoot()]`
Arity	`#i:arity=n,m`	`$ni[arity(n,m)]`
Token arity	`#i:tokenArity=n,m`	`$ni[tokenArity(n,m)]`

Meta annotations `meta::namespace:name="value"` are mapped to the custom node type `meta(namespace:name="value")`.

C SQL Schema of the Corpus Data Model

Reproduced below are the table definitions of the SQL schema of the corpus data model defined in section 2 along with the modifications made in section 4.

Listing 4: Table definitions for the SQL schema of the corpus data model.

```
CREATE TABLE corpus
(
      id          numeric(38) PRIMARY KEY,
      name        varchar(100) NOT NULL,  -- unused
      type        varchar(100) NOT NULL,  -- unused
      version     varchar(100),           -- unused
      pre         numeric(38) NOT NULL UNIQUE,
      post        numeric(38) NOT NULL UNIQUE,
      top_level   boolean NOT NULL        -- see section 4.5
);

CREATE TABLE corpus_annotation
(
      corpus_ref  numeric(38) NOT NULL REFERENCES corpus (id),
      namespace   varchar(100),
      name        varchar(1000) NOT NULL,
      value       varchar(2000),
      UNIQUE (corpus_ref, namespace, name)
);

CREATE TABLE text
(
      id          numeric(38) PRIMARY KEY,
      name        varchar(1000),    -- unused
      text        text              -- unused
);

CREATE TABLE node
(
      id          numeric(38) PRIMARY KEY,
      text_ref    numeric(38) NOT NULL REFERENCES text (id),
      corpus_ref  numeric(38) NOT NULL REFERENCES corpus (id),
      namespace   varchar(100),
      name        varchar(100) NOT NULL,
      "left"      integer NOT NULL,
      "right"     integer NOT NULL,
      token_index integer,
      continuous  boolean,
      span        varchar(2000),
      toplevel_corpus numeric(38) NOT NULL REFERENCES corpus (id),      -- see section 4.5
      left_token  integer NULL,                                         -- see section 4.3.6
      right_token integer NULL
);
```

Listing 4: Table definitions for the SQL schema of the corpus data model (continued).

```sql
CREATE TABLE node_annotation
(
      node_ref    numeric(38) REFERENCES node (id),
      namespace   varchar(150),
      name        varchar(150) NOT NULL,
      value       varchar(1500),
      UNIQUE (node_ref, namespace, name)
);

CREATE TABLE rank
(
      pre               numeric(38) PRIMARY KEY,
      post              numeric(38) NOT NULL UNIQUE,
      node_ref          numeric(38) NOT NULL REFERENCES node (id),
      component_ref     numeric(38) NOT NULL REFERENCES component (id),
      parent            numeric(38) NULL REFERENCES rank (pre),
      root              boolean,                                -- see section 4.3.8
      level             numeric(38) NOT NULL                    -- see section 4.3.7.4
);

CREATE TABLE component
(
      id          numeric(38) PRIMARY KEY,
      type        char(1),
      namespace   varchar(255),
      name        varchar(255)
);

CREATE TABLE edge_annotation
(
      rank_ref    numeric(38) REFERENCES rank (pre),
      namespace   varchar(150),
      name        varchar(150) NOT NULL,
      value       varchar(1500),
      UNIQUE (rank_ref, namespace, name)
);
```

D Experimental Setup

D.1 Test queries

Table 23 lists the test queries used in section 6. They can be characterized by the number of search terms and the type and number of linguistic constraints referring to these terms (Table 18).

Table 18: Number of search terms and operations per query. Coverage and precedence are text operations; dominance and pointing relations are graph operations.

Query	Solutions	Search terms	Text operations		Graph operations	
			direct	indirect	direct	indirect
1	13	3	2			
2	26	3	1		1	
3	1	2		1		1
4	8	4	2			1
5	156	4	3		1	
6	976	3	1	1	1	
7	297	3		1	2	
8	131	4			3	
9	666	4		2	3	
10	169	2	1			
11	860	2	1			
12	3929	3	2			

D.2 The Tiger corpus

The PAULA representation of the Tiger corpus is about 500 MB large. It contains annotations over a little more than 625000 tokens from 1558 texts. Table 19 lists statistical information about the corpus and Table 20 the row count for each table in the corpus data model.

Table 19: General information about the Tiger corpus.

Nodes	889476
Tokens	625778
Roots	155663
Edges	1556468
Node duplicates in rank	826008
Unconnected nodes	86918
Average number of nodes in a component	10.8
Average number of spans in a text	571

Tiger contains annotation values for four node annotation names and one edge annotation name. These are listed in Table 21 along with the number of their distinct values. Table 22 lists annotation values that are not unique to one node annotation name.

Table 20: Number of tuples for each table.

Table	**Tuples**
text	1558
node	889476
node_annotation	2141032
rank	1715585
component	159016
edge_annotation	1556468
facts	4073090

Table 21: Number of distinct values for each node and edge annotation name.

Annotation name	**Distinct values**
node annotations	
cat	26
pos	54
morph	259
lemma	52174
edge annotations	
func	44

Table 22: Common annotation values for node annotations.

Name	**Value**	**Occurrences**
lemma annotations		
cat	--	2
cat	AA	80
cat	CO	261
cat	CS	4143
cat	PP	64144
cat	S	50846
morph	--	216524
morph	Gen	8
pos	PDS	2262
cat annotations		
lemma	--	84787
lemma	AA	1
lemma	CO	5
lemma	CS	2
lemma	PP	2
lemma	S	6
morph	--	216524
pos annotations		
lemma	PDS	110
morph annotations		
cat	--	2
lemma	--	84787
lemma	Gen	12

D.3 Test system

All queries were performed on a 2.8 GHz Intel Core 2 Duo processor with 2 GB RAM running a standard Ubuntu 9.10 Linux kernel and PostgreSQL 8.4.3. For the $COUNT$ query function the Annis client ran on the same machine; for the $ANNOTATE$ and $MATRIX$ functions we used a dedicated remote PostgreSQL host to eliminate interference of the Annis Java process with the Linux disk cache.

D.4 PostgreSQL configuration

The default configuration of PostgreSQL uses system resources very sparsely. To improve the performance of Annis it is necessary to change the settings listed in Listing 5 in the PostgreSQL configuration file postgresql.conf. Most of the options shown in the excerpt below are commented out in postgresql.conf. This means that PostgreSQL will use the default value for this option, i.e. the value as it appears in the default postgresql.conf file. More information on these settings can be found in the PostgreSQL manual [24].[24]

Listing 5: PostgreSQL configuration used throughout the experiments in section 6.

```
max_connections = 10
effective_cache_size = 1536MB # 75% of RAM; estimated size of OS disk cache
shared_buffers = 512MB        # 25% of RAM; memory shared across all sessions
work_mem = 128MB              # RAM / (2 x max_connections); memory used for *one*
                              # sort, aggregate or hash operation inside a query plan
maintenance_work_mem = 512MB  # RAM for maintenance operations during corpus import
checkpoint_segments = 30      # also affects corpus import
```

D.5 Configuration of system resources

PostgreSQL needs to access large areas of continuous RAM which can easily exceed the maximum size allowed by the operating system. PostgreSQL will check the OS resource settings during startup and exit with an error if they are not adequate.

Reproduced below are the commands to change the resource settings on Linux and OS X. More information can be found in the PostgreSQL manual.[25]

On Linux:

```
sysctl -w kernel.shmmax=536870912        # bytes; corresponds to 512MB
```

This command takes effect immediately. To make the change permanent across system reboots, add it to the file /etc/sysctl.conf.

On Mac OS X:

```
sysctl -w kern.sysv.shmmax=536870912     # bytes; corresponds to 512MB
sysctl -w kern.sysv.shmall=131072        # measured in 4 kB pages
```

These commands have to be added to the file /etc/sysctl.conf and OS X has to be rebooted for the changes to take effect.

---

[24] Sections 18.4. Resource Consumption, 18.5. Write Ahead Log, 18.6. Query Planning and 28.4. WAL Configuration.

[25] Section 17.4. Managing Kernel Resources.

Table 23: Test queries used in the experiments.

	Query	Hits	Description in German	Example
1	`pos="KOUS" & "man" & "sich" &` `#1 . #2 & #2 . #3`	13	Nebensätze mit Subjekt „man“ und einem Reflexivum	weil man sich, ob man sich
2	`cat="S" & pos="PTKVZ" & pos="VVFIN" &` `#1 > #3 & #1 _l_ #2`	26	Verbzweit-Sätze mit vorangestellter Verbpartikel	Verloren ging dabei ..., Fest steht, dass ...
3	`"desto" & "desto" & #1 $* #2 & #1 .* #2`	1	Je-desto-Satz mit zwei Desto-Teilsätzen	Je ärmer, desto mehr Einkommen geht fürs Essen drauf – desto höher der relative Beitrag zur GAP.

4	`/[Jj]e/ & "desto" & #1 $* #2 &`
	`morph="Comp" & morph="Comp" &`
	`#1 . #3 & #2 . #4`
		8	Je-desto-Sätze mit den dazu gehörigen komparativen Adjektiven	Je höher man komme, desto gefährlicher würden die Felsvorsprünge

5	`cat="S" & pos="VVFIN" & cat & cat &`
	`#1 >[func="HD"] #2 &`
	`#1 _l_ #3 &`
	`#3 . #4 & #4 . #2`
		156	Mehrfache Vorfeldbesetzung	Negativ auf den Gewinn wirkten sich vor allem Wechselkursschwankungen aus.

6	`tok & cat="VP" & #1 . #2 &` `tok & #2 _i_ #3 & #1 $ #3`	976	VP wird durch ein Token von der übergeordneter Phrase unterbrochen	Fest stehe, und da meint Bereketi dem Parlament durchaus vorgreifen zu können, dass ...

7	`cat="S" & cat="NP" & cat="NP" &`
	`#1 >[func="OA"] #2 & #1 >[func="SB"] #3 &`
	`#2 .* #3`
		297	Sätze, in denen das Objekt vor dem Subjekt steht	Nach Berichten von Augenzeugen verübte den Anschlag ein Selbstmordkommando.

8	`pos="NE" & cat="S" & pos="PRELS" & pos="VVFIN" &`
	`#2 >[func="HD"] #4 &`
	`#1 $ #2 & #3 $ #4`
		131	Eigennamen, die durch Relativsätze erweitert werden, mit den dazu gehörigen Nebensatzvollverben	Diese georgische Provinz strebt die Vereinigung mit Nordossetien an, das zu Russland gehört.

9	`cat="S" & node & pos="VVFIN" & node &`
	`#1 >[func="OA"] #2 & #1 > #3 & #1 >[func="SB"] #4 &`
	`#2 .* #3 & #3 .* #4`
		666	Sätze, in denen das direkte Objekt vor und das Subjekt nach dem Verb steht	Was aber bezweckten die Sieger?

10	`lemma=/.+[^aeiouäöü]chen/ & pos="NN" &` `#1 _=_ #2`	169	Verkleinerungsformen	Bändchen, Kästchen, Hintertürchen
11	`pos=/N.*/ & /[12][09][0-9][0-9]/ &` `#1 . #2`	860	Nomina einschließlich Eigennamen vor Jahreszahlen	Oktober 1970, Wunderlich/Fabri 1995
12	`lemma=/[^äöü]+/ & tok=/.+[äöü].+/ & pos="NN" &` `#1 _=_ #2 & #2 _=_ #3`	3929	Nomina mit Umlaut in der Pluralform	Züge, Fragesätze, Entwürfe

References

[1] ARFF Format Definition. `http://weka.wikispaces.com/ARFF+%28book+version%29`, Retrieved on 2009/10/16.

[2] DeutschDiachronDigital. `http://www.deutschdiachrondigital.de/`.

[3] The Apache Xerces Project. `http://xerces.apache.org/`.

[4] Srini Acharya, Cesar Galindo-Legaria, Milind Joshi, Babu Krishnaswamy, Stefano Stefani, and Pawel Terlecki. Filtered indices and their use in flexible schema scenarios. In *Proceedings of the International Conference on Data Engineering*, pages 1259–1266, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[5] Shurug Al-Khalifa, H. V. Jagadish, Nick Koudas, Jignesh M. Patel, Divesh Srivastava, and Yuqing Wu. Structural joins: A Primitive for Efficient XML Query Pattern Matching. In *Proceedings of the International Conference on Data Engineering*, pages 141–154. IEEE Computer Society Press; 1998, 2002.

[6] W. Alink, R. Bhoedjang, A. P. de Vries, and P. A. Boncz. Efficient XQuery Support For Stand-Off Annotation. In *Proceedings of International Workshop on XQuery Implementation, Experience and Perspectives 2006 (3)*, pages 1 – 6. ACM Press, 2006.

[7] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, page 490. ACM, 2006.

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

[9] J. Carletta, S. Evert, U. Heid, and J. Kilgour. The NITE XML toolkit: data model and query language. *Language resources and evaluation*, 39(4):313–334, 2005.

[10] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient Structural Joins on Indexed XML Documents. In *Proceedings of the 28th international conference on Very Large Data Bases*, page 274. VLDB Endowment, 2002.

[11] B.F. Cooper, N. Sample, M.J. Franklin, G.R. Hjaltason, and M. Shadmon. A Fast Index for Semistructured Data. In *Proceedings of the 27th VLDB Conference*, pages 341–350, 2001.

[12] Stefan Evert and Holger Voormann. NITE Query Language. *To appear*, 2002.

[13] S. Gorn. Explicit Definitions and Linguistic Dominoes. *Systems and Computer Science*, page 77, 1965.

[14] Michael Götze and Viktor Rosenfeld. *ANNIS-QL 1.0 Spezifikation*. SFB 632, draft edition, May 2008.

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

[16] Torsten Grust, Jan Rittinger, and Jens Teubner. Why Off-the-Shelf RDBMSs are Better at XPath Than You Might Expect. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, page 958. ACM, 2007.

[17] Torsten Grust, Marice van Keulen, Jens, and Teubner. Staircase join: Teach a Relational DBMS to Watch its (Axis) Steps. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, page 535. VLDB Endowment, 2003.

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

[19] Esther König, Wolfgang Lezius, and Holger Voormann. TIGERSearch User's Manual. *IMS, University of Stuttgart, Stuttgart*, 2003.

[20] Rajesekar Krishnamurthy, Raghav Kaushik, and Jeffrey F. Naughton. XML-to-SQL Query Translation Literature: The State of the Art and Open Problems. *Lecture notes in computer science*, pages 1–18, 2003.

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

[22] T. Marek, J. Lundborg, and M. Volk. Extending the TIGER Query Language with Universal Quantification. In *Proceeding of KONVENS*, pages 3–14, 2008.

[23] Sabine Mayer. Enhancing the Tree Awareness of a Relational DBMS: Adding Staircase Join to PostgreSQL. Master's thesis, Universitat Konstanz, 2004.

[24] PostgreSQL Global Development Group. PostgreSQL 8.4 Manual. `http://www.postgresql.org/docs/8.4/interactive/index.html`.

[25] Universität Potsdam. Annis 2. `http://www.sfb632.uni-potsdam.de/~d1/annis`.

[26] P. Seshadri and A. Swami. Generalized Partial Indexes. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 420–427, 1995.

[27] M. Stonebraker. The Case for Partial Indexes. *ACM Sigmod Record*, 18(4):11, 1989.

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

[29] Thorsten Vitt. DDDquery: Anfragen an komplexe Korpora. Diplomarbeit, Humboldt-Universität zu Berlin. `https://www.informatik.hu-berlin.de/forschung/gebiete/wbi/teaching/studienDiplomArbeiten/finished/2005/vitt_diplomarbeit.pdf`, 2005.

[30] M. Volk, J. Lundborg, and M. Mettler. A Search Tool for Parallel Treebanks. In *Proceedings of the Linguistic Annotation Workshop*, pages 85–92. Association for Computational Linguistics, 2007.

[31] Chun Zhang, Jeffrey F. Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, page 436. ACM, 2001.

[32] Florian Zipser. Entwicklung eines Konverterframeworks für linguistisch annotierte Daten auf Basis eines gemeinsamen (Meta-)modells. Diplomarbeit, Humboldt-Universität zu Berlin. `http://www.linguistik.hu-berlin.de/institut/professuren/korpuslinguistik/mitarbeiter-innen/florian/pdf/diplomarbeit.pdf`, 2009.

[33] Humboldt-Universität zu Berlin. SaltNPepper Wiki. `https://korpling.german.hu-berlin.de/trac/saltnpepper`.