Binary indexes for optimising corpus queries

Peter Ljunglöf[1,2] and **Nicholas Smallbone**[1,2] and **Mijo Thoresson**[1] and **Victor Salomonsson**[1]
[1]Computer Science and Engineering and [2]Språkbanken Text
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden
peter.ljunglof@cse.gu.se, nicsma@chalmers.se,
mijot@student.chalmers.se, vicsal@student.chalmers.se

Abstract

To be able to search for patterns in annotated text corpora is crucial for many different research disciplines. However, searching for complex patterns in large corpora can take long time – sometimes several minutes or even hours.

We investigate how inverted indexes can be used for efficient searching in large annotated corpora, and in particular *binary indexes*. We show how corpus queries are translated into lookups in unary and binary inverted indexes, and give efficient strategies for combining the results using efficient set operations. In addition we discuss how to make use of binary indexes for more complex query types.

1 Introduction

Annotated text corpora are used for research in humanities and social sciences to answer questions such as: How has the use of a certain word or phrase changed over time? What grammatical constructions are the most difficult for non-native speakers? How does politicians' rhetoric around migration vary by political party and audience?

To answer such questions, specialised corpus search tools are used. As corpora can be extremely large (for example, the *News on the Web* corpus[1] consists of 18.9 billion tokens), and queries can be complex (mentioning linear order, syntactic dependencies, logical connectives and more), it is difficult to execute the queries efficiently. Unfortunately, existing search tools are either restricted and cannot express all the kinds of constraints we want, or they are inefficient on large corpora, with queries taking minutes or even hours to complete.

This paper presents new techniques for answering corpus queries more efficiently. We build on the standard technique of an *inverted index*, which can be used to find all corpus positions where a given token occurs. We introduce a new type of index that we call a *binary index*, which is an inverted index over pairs of tokens.[2] This new type of index can sometimes reduce query times by several orders of magnitude. We present a new corpus search algorithm that can answer queries more efficiently by combining lookups from *multiple indexes*. Finally, we show how to extend our algorithm to handle more types of corpus queries, by reducing complex queries into simpler ones. Our prototype tool performs well and is available as open source.[3]

2 Background

In this section we describe how corpus engines work and what kind of problems they face.

2.1 Corpus query languages

There are two main approaches for how to formulate search queries in text corpora – *linear* vs. *structured* query languages. Linear queries are easier to make efficient (so better suited for large corpora), while structured queries are more powerful (but on the other hand slower to execute).

In a *linear query model* you can formulate queries about annotated tokens, and their relationship with neighbouring tokens. The model usually supports referring to immediate neighbours and to neighbours some tokens away. However, it is more difficult to formulate queries about long-distance dependencies or syntactic structure. Variants include the IMS Corpus Query Language (Evert and Hardie, 2011) and the Poliqarp Query Language (Bingel and Diewald, 2015).

With a *structured query model* you can search for long-distance dependencies or syntactic structure such as nested phrases, anaphoric references or discontinuous multi-word entities. The model can be tree-based (e.g., Ghodke and Bird, 2012;

[1]NOW, https://www.english-corpora.org/now/

[2]Another possible term is *bigram index*, but we choose not to use this because a bigram usually refers to adjacent tokens, but our binary indexes can span arbitrary distances.

[3]https://github.com/heatherleaf/korpsearch

0	1	2	3	4	5	6	7
The	*large*	*houses*	*of*	*the*	*middle*	*class*	*were*
DT	JJ	NN	IN	DT	JJ	NN	VB

8	9	10	11	12	13	14	15
divided	*into*	*tenements*	*to*	*house*	*the*	*swarming*	*population*
VB	IN	NN	IN	VB	DT	JJ	NN

Figure 1: An example sentence from the British National Corpus, annotated with parts of speech.

2(a). [*pos*=NN] [*word*=TO] [*word*=HOUSE, *pos*=VB]

2(b). [*pos*=NN] [*word*=TO] [*word*=HOUSE, *pos*≠VB]

2(c). [*word*=THE] [*pos*=JJ] [*pos*=NN]

2(d). [*word*=THE] ([*pos*=JJ] | [*pos*=NN]) [*pos*=NN]

2(e). ([*pos*=JJ] | [*pos*=IN]) [*word*=H[AEIOUY]*SE .*]

Figure 2: Example corpus queries.

Robie et al., 2017) or graph-based (e.g., Krause and Zeldes, 2016; Luotolahti et al., 2017; Kleiweg and van Noord, 2020), and is often tailor-made for a certain type of structured annotation, such as UD treebanks (de Marneffe et al., 2021).

In this paper we focus on linear query languages, and leave long-distance dependencies and syntactic structure as future work.

2.2 Corpus search engines

Corpus search engines can be divided into two main approaches: the *inverted index* approach, or the *database* approach.

Engines such as Corpus Workbench (Evert and Hardie, 2011) and Corpuscle (Meurer, 2020) build one or more inverted indexes from the corpus, which then are used to optimise search. They analyse a given query to find out which index to use, and uses the index to find a set of potential candidates. The set is then filtered by testing each candidate if it matches the query or not. Engines can be more or less intelligent when they decide which index to use – Corpus Workbench always uses the index corresponding to the first token in the query, while, e.g., Corpuscle tries to find an optimal cut in a finite automaton to decide which index to start from.

The second approach is to translate the corpus into a relational database. E.g., Davies (2005) transforms consecutive tokens in a corpus into a database of n-grams, AlpinoGraph (Kleiweg and van Noord, 2020) compiles treebanks into graphs stored in an SQL database, Krill (Diewald and Margaretha, 2016) uses the Apache Lucene information retrieval engine as a backbone, while LiRI (Schaber et al., 2023) converts the corpus and its annotations into tables designed to make use of the full-text search capabilities of PostgreSQL (2024, ch. 12).

In this paper we use the first approach and build the indexes ourselves. However, note that the second approach indirectly also uses indexes because they are automatically created by the underlying database engine.

2.3 Inverted indexes

Our implementation of inverted indexes are related to *suffix arrays* (Manber and Myers, 1993), which are efficient indexes for efficient full-text search in almost-constant time. Suffix arrays and its descendant algorithms are used in *information retrieval*, and the main difference to our approach is that information retrieval research focuses on pure text searches – i.e., finding substrings or patterns in plain text. As a contrast we have to be able to handle annotations on different levels, and not just text as a stream of characters.

2.4 Drawbacks of existing approaches

As far as we know, existing approaches do not combine multiple search indexes. When given a complex query, they usually use one of the indexes to get a collection of potential search results, and then filter the results one by one, by testing if they match the query.

In addition, no existing corpus engine uses binary search indexes, and as we show in section 5 they can drastically improve some queries.

3 Definitions and semantics

3.1 Annotated corpora

For the purposes of this paper, an annotated corpus is a collection of *texts*. Each text consists of *sentences* which in turn consist of *tokens*. Each token is annotated with a number of *attributes*, such as *word* (surface form), *lemma*, *pos* (part of speech), etc., where each attribute has one single value.

This definition of corpus is restricted – currently we cannot handle multi-token annotations, set-valued attributes, structural attributes, or empty tokens, to name just a few possibilities.

Formally, a corpus C is a sequence of tokens $C[0]\ C[1] \ldots C[n{-}2]\ C[n{-}1]$, where each token is an attribute-value mapping. We write $C[i].\mathit{pos}$ for the value of attribute *pos* at position i.

Figure 1 shows an example sentence taken from the British National Corpus (BNC), annotated with just word form (*word*) and part of speech (*pos*).

Note that we assume for now that the corpus is not divided into larger structures, such as phrases, sentences, paragraphs or texts. This will be discussed later in section 6.

3.2 Queries

In the next few sections, we use a restricted version of CQL (the Corpus Query Language, see section 2.2.3 in Evert and Hardie, 2011). Sections 7–8 then show how to lift some of the restrictions.

A query is of the form [*literal**]+, where a literal is either *attr*=*value* or *attr*≠*value*. The example query in figure 2(a) searches for sentences which contain a noun, followed by the word "*to*", followed by the word "*house*" tagged as a verb, whereas query 2(b) requires that the word "*house*" is not a verb. Query 2(c) is very generic and matches all words "*the*" followed by an adjective and a noun.

The remaining two queries use features not present in the restricted query language. Query 2(d) uses disjunction, so that the middle word may be an adjective or a noun, and query 2(e) uses a regular expression. In section 7, we extend the search algorithm to handle both these kinds of queries.

3.3 Query semantics

The token [*word*=HOUSE, *pos*=VB] in query 2(a) occurs 2 tokens after the first query token; we say that it has *relative position* 2. Using relative positions, we can write query 2(a) more formally as

$$[\mathit{pos}^{@0}{=}\text{NN}] \wedge [\mathit{word}^{@1}{=}\text{TO}] \\ \wedge [\mathit{word}^{@2}{=}\text{HOUSE}] \wedge [\mathit{pos}^{@2}{=}\text{VB}]$$

where $[\mathit{word}^{@2}{=}\text{HOUSE}]$ means that the word at relative position 2 is "*house*". Now we can define the semantics of a literal l at relative position k as the set of all positions p such that l is true at position $p{+}k$:

$$[\mathit{attr}^{@k}{=}\mathit{val}] \ \equiv\ \{\ p \mid C[p{+}k].\mathit{attr} = \mathit{val}\ \}$$

We call this set a *query set* and we write it $\{\mathit{attr}^{@k}{=}\mathit{val}\}$. The semantics of a combined query is then the intersection of the query sets for each of the literals in the query:

$$\{\mathit{pos}^{@0}{=}\text{NN}\} \cap \{\mathit{word}^{@1}{=}\text{TO}\} \\ \cap \{\mathit{word}^{@2}{=}\text{HOUSE}\} \cap \{\mathit{pos}^{@2}{=}\text{VB}\}$$

If a literal is negated $\{\mathit{attr}^{@k}{\neq}\mathit{val}\}$, we instead take the set difference with the corresponding positive literal. The semantics of query 2(b) then becomes:

$$\{\mathit{pos}^{@0}{=}\text{NN}\} \cap \{\mathit{word}^{@1}{=}\text{TO}\} \\ \cap \{\mathit{word}^{@2}{=}\text{HOUSE}\} \setminus \{\mathit{pos}^{@2}{=}\text{VB}\}$$

4 Efficient inverted indexes

In this section we describe how we build search indexes from a corpus to facilitate efficient search. As mentioned in section 2.2, the idea of using inverted indexes is not new, in fact large-scale corpus search engines compile the corpus into some kind of search indexes. What we present at first is a fairly standard inverted index. But afterward we move to what is new: how to make use of more than one search index when executing a complex query, and binary indexes.

Each annotation attribute (*pos*, *word*, etc.) is precompiled into an inverted index of corpus positions. This index is inspired by suffix arrays (Manber and Myers, 1993), in that we do not have to store the values in the index – it is just a large array of corpus positions. The array is sorted alphabetically on the attribute value at the given position. When there are many tokens with the same attribute value, these positions are in increasing order.

For example, assume that the example sentence in figure 1 is our whole corpus. Then the index for the *pos* attribute will be the following array of positions:

$$[\underbrace{0,4,13}_{\text{DT}},\underbrace{3,9,11}_{\text{IN}},\underbrace{1,5,14}_{\text{JJ}},\underbrace{2,6,10,15}_{\text{NN}},\underbrace{7,8,12}_{\text{VB}}]$$

This array is sorted alphabetically on the *pos* values: [0, 4, 13] are the determiners (DT), [3, 9, 11] are the prepositions (IN), etc. Furthermore, each group of positions for the same value is in increasing order.

So a search index is simply a large array of integers, which can be stored as a memory-mapped binary file of fixed-size integers for fast access.

4.1 Searching an inverted index

To search for a value in an index we can do two very efficient binary searches – one that finds the first matching value and another that finds the last match. If we search for NN (a noun) in the example index, these searches return 6 and 9, which are the start and end indices for the sublist [2, 6, 10, 15], which contain all the corpus positions for NN.

Now, to execute the query $\{pos^{@k}\text{=NN}\}$, we search for NN in the index, and then subtract k from all matching positions. But for efficiency we instead just record the start and end indices (6 and 9) and the relative position k, using which we can easily recover all matching positions.

So the result of an index lookup can be stored as a tuple (i, j, k) where i and j denote the relevant span in the search index, and k is the relative position. In particular, we do not need to load the result set into memory.

Note that we cannot use this simple approach if we have a negative literal [*attr*≠*val*], because inverted indexes do not store complement sets. Instead we have to calculate the set difference, which is described in the next section.

4.2 Computing the result of a query

To execute a complex query, we look up each literal to get its query set, as described in the last section. Then we translate the query into a set theory expresssion as described in section 3.3, and then just evaluate the expression, using set intersection, difference and union to find the final result.

4.3 Computing query sets

As described in section 4.1 the initial query sets are just pointers into the inverted indexes. But when performing the set operations we have to build the resulting sets.

The query sets are stored as sorted arrays, and there are simple and efficient algorithms for computing the intersection, difference and union. The results are also sorted arrays themselves, so we can continue using these algorithms to compute the final result. Depending on the relative sizes of the sets we use one of the following two algorithms:

Merge The default is to use a *merging* strategy: Iterate through both sets in parallel, adding elements to the result set. If the sizes of the two sets are n and m, this algorithm has complexity $O(n + m)$.

Filter If one set is much larger than the other, we can use a *filtering* strategy: Iterate through each element of the smaller set, and test if it is also in the larger set using binary search. The complexity of this algorithm is $O(n \log m)$, where n is the size of the smaller set. Note that this strategy cannot be used for computing the union.

4.4 Deciding the order of the set operations

If we have more than two query sets, we have to decide in which order to perform the set operations. It is not always the case that starting from the left-most token is the best in all circumstances – the order can have a huge difference.

A heuristic that works well for intersection and difference is to start from the smallest sets and leave the largest until later. This is because the result set will never be larger than the original sets, and then we avoid doing duplicate work.

Set union is different, because the result set will be increasing. This case is discussed in section 7.1.

4.5 Example

We tested this algorithm on the 112 million token British National Corpus (BNC).[4] The resulting query sets for query 2(a) are as follows:

$\{pos^{@0}\text{=NN}\}$	$\rightarrow$	26 M results
$\{word^{@1}\text{=TO}\}$	$\rightarrow$	2.6 M results
$\{word^{@2}\text{=HOUSE}\}$	$\rightarrow$	33 k results
$\{pos^{@2}\text{=VB}\}$	$\rightarrow$	18 M results

We start by intersecting the smallest query sets, $\{word^{@2}\text{=HOUSE}\}$ and $\{word^{@1}\text{=TO}\}$, which gives 421 results. Then we intersect with $\{pos^{@2}\text{=VB}\}$ and finally with $\{pos^{@0}\text{=NN}\}$, in the end finding 158 search results.

Intersection uses the *filtering strategy* from section 4.3, which only needs to iterate through the smallest index. In the first step it iterates through $\{lemma^{@2}\text{=HOUSE}\}$, and in the second step it only iterates through the 421 intermediate results. Recall also that the initial query sets are not loaded into memory, but are stored as a tuple as described in section 4.1. Due to these optimisations, the query runs quickly, in about 0.3s on an ordinary laptop.

To calculate query 2(b) we use the same initial query sets. But instead of intersecting with $\{pos^{@2}\text{=VB}\}$ we take the set difference, and in the end we find that there are 38 search results.

4.6 When are unary indexes not enough?

However, there are still cases where using the simple search indexes are inefficient. Consider the very general query 2(c), which is rewritten to this:

$$\{word^{@0}\text{=THE}\} \cap \{pos^{@1}\text{=JJ}\} \cap \{pos^{@2}\text{=NN}\}$$

Each of these literals results in a huge set:

[4]BNC, http://www.natcorp.ox.ac.uk/

$\{word^{@0}=\text{THE}\}$	$\rightarrow$	5 M results
$\{pos^{@1}=\text{JJ}\}$	$\rightarrow$	18 M results
$\{pos^{@2}=\text{NN}\}$	$\rightarrow$	26 M results

So the intersections become slower (about 20 times slower than the previous example, taking about 6s). The first intersection gives 1.5 M results, and the second one results in 1.1 M final results.

To solve this we now introduce *binary indexes*.

5 Binary query indexes

Formally, a unary query index [*a*] can be seen as a function from values to query sets:

$$[a] \quad \equiv \quad \lambda v \rightarrow \{a^{@0}{=}v\}$$

Similarly a *binary query index* can be viewed as a function from pairs of values to query sets:

$$[a]\ [b] \quad \equiv \quad \lambda v, w \rightarrow \{a^{@0}{=}v\} \cap \{b^{@1}{=}w\}$$
$$[a]\ []\ [b] \quad \equiv \quad \lambda v, w \rightarrow \{a^{@0}{=}v\} \cap \{b^{@2}{=}w\}$$

(similar for [*a*] [] [] [*b*], etc.)

For example, an index [*word*] [] [*pos*] can answer queries such as [*word*=THE] [] [*pos*=NN]. These binary indexes can be compiled and searched in a similar way to the unary indexes.

5.1 Searching using binary indexes

Now we can decompose a complex query into a composition of binary indexes. E.g., if we have computed binary indexes for adjacent tokens ([*a*][*b*]) and for tokens with a gap ([*a*][][*b*]), a query with three adjacent tokens, $[t_1][t_2][t_3]$, is equivalent to any of the following binary index searches:

$$[t_1]\ [t_2] \ \cap\ [t_2]\ [t_3]^{@1}$$
$$[t_1]\ [t_2] \ \cap\ [t_1]\ []\ [t_3]$$
$$[t_1]\ []\ [t_3] \ \cap\ [t_2]\ [t_3]^{@1}$$

Exactly which of these is the most efficient depends on the sizes of the resulting query sets. In this case, we calculate all three query sets and then take the intersection of the two smallest.

5.2 Results using binary indexes

Using the same example as in section 4.6, we search in the following binary indexes, instead of the unary indexes we tried before:

[*word*=THE] [*pos*=JJ]	$\rightarrow$	1.4 M results
[*word*=THE] [] [*pos*=NN]	$\rightarrow$	1.7 M results
[*pos*=JJ] [*pos*=NN]	$\rightarrow$	6.7 M results

Now we can intersect the two smaller sets:

$$[word{=}\text{THE}]\ [pos{=}\text{JJ}]$$
$$\cap\ [word{=}\text{THE}]\ []\ [pos{=}\text{NN}]$$

This intersection gives 1.1 M results, and we do not have to use the other indexes: by set theory, the intersection above describes the same set as the query, so we have the correct result already.

The total query time is reduced from 6s to 0.4s. (On the same query, Corpus Workbench takes 10s.)

5.3 Search heuristics for binary indexes

Finally we are ready to describe the heuristics we use to decide in which order we perform intersections and set difference:

1. Infer which binary indexes are relevant;
2. Perform all relevant binary index lookups;
3. If some token is not covered by a binary index, look it up in the unary index;
4. Perform intersections starting from the smallest set, until the whole query is covered;
5. If the query contains negative literals, look up the value in the unary index, and calculate the set difference instead of the intersection.

5.4 How many binary indexes are needed?

Each binary index is as large as a unary index, and there are many possible binary indexes. If we have n different attributes (and therefore n unary indexes), then there are n^2 possible binary indexes per relative distance. So there are n^2 [*a*][*b*] indexes, and n^2 [*a*][][*b*] indexes, etc. This is potentially very many indexes that take up a lot of space.

But we do not have to build all these indexes. Note that any query with k adjacent tokens can be simplified into a conjunction of $k-1$ lookups in [*a*][*b*] indexes, as shown in section 5.1. Therefore it should be enough to only build n^2 *bigram* indexes. However, as seen in 5.1, it is often useful to also build the n^2 [*a*][][*b*] indexes, because then we get several different ways of searching to find the most optimal intersection order. But it is usually not worth the trouble to build indexes with longer relative distances, such as [*a*][][][][*b*].[5]

Also note that if a binary index is missing, we can simply fall back to searching in two unary indexes instead, as in section 4. This means that we can focus on building binary indexes only for the kinds of queries where they have the greatest impact.

[5]The one exception is if the query itself has a longer gap, such as $[t_1][][][][t_2]$, then we have to resort to searching in unary indexes instead.

5.5 Reducing the size of binary indexes

Still, each binary index is as large as a unary index, and storing up to $2n^2$ binary indexes can use up quite a lot of space. So can we reduce their size in any way?

If a query uses a literal that is uncommon in the corpus (e.g., [*word*=TURTLE] only occurs 166 times in BNC), there is no need to use binary indexes for that query, since the unary index will already return a small query set. Therefore, an optimisation is to only add a new index instance (*v*, *w*) to the index [*a*][*b*], if the corresponding unary instances *v* and *w* are common enough in [*a*] and [*b*] respectively. When we execute a query, we then need to check which literals are uncommon, and exclude the use of binary indexes for those literals.

For example, in the BNC each full (unary and binary) index uses around 400 MB. If we only include pairs where both words occur at least 20,000 times each, the binary indexes are reduced to around half their size.

6 Sentences and hierarchical structures

The corpus is encoded as a sequence of tokens, and a sentence starts directly after the previous one ends. So how can we ensure that we don't match sentence borders? E.g., we don't want query 2(c) to match a sentence that ends in "the first" where the next sentence starts with an arbitrary noun.

To solve this we encode the start of a sentence as an attribute of its own. So we build an index [*s*] which has a special value (say •) only for the tokens that start a sentence. Our example query is then translated to:

$$[word^{@0}{=}\text{THE}] \wedge [s^{@1}{\neq}\bullet] \wedge [pos^{@1}{=}\text{JJ}] \\ \wedge [s^{@2}{\neq}\bullet] \wedge [pos^{@2}{=}\text{NN}]$$

6.1 Sentence borders and binary indexes

To handle sentence borders and binary indexes we can incorporate the literals [$s^{@1}{\neq}\bullet$] in our binary indexes. So their meaning is actually:

$$\begin{aligned} [a]\,[b] &\equiv \lambda v, w \to \{a^{@0}{=}v\} \cap \{b^{@1}{=}w\} \\ &\qquad \cap \{s^{@1}{\neq}\bullet\} \\ [a]\,[\,]\,[b] &\equiv \lambda v, w \to \{a^{@0}{=}v\} \cap \{b^{@2}{=}w\} \\ &\qquad \cap \{s^{@1}{\neq}\bullet\} \cap \{s^{@2}{\neq}\bullet\} \end{aligned}$$

That is, the indexes exclude matches which cross a sentence border. Though this perhaps looks complicated, it can be generated automatically, and keeps query execution simple. Our example query 2(c) can still be translated to searches in the following three binary indexes:

[*word*][*pos*], [*word*][][*pos*], and [*pos*][*pos*]

And just as in section 5.2, we only have to intersect the two smallest query sets because the final query set is subsumed by the intersection.

7 Extending the query language

Here we show how we handle more expressive queries than the very simple ones described earlier.

7.1 Disjunctive queries

CQL supports disjunction in queries. For example, query 2(d) is of the form $A(B|C)D$, where A searches for the word "the", B an adjective, C a noun, and D a noun.

If we use only unary indexes each literal corresponds to a index lookup, so query 2(d) results in calculating the set $A \cap (B \cup C) \cap D$. In order to make use of the binary indexes, we expand out the disjunction into two *strands*:

$$\begin{aligned} ABD &= [word{=}\text{THE}]\ [pos{=}\text{JJ}]\ [pos{=}\text{NN}] \\ ACD &= [word{=}\text{THE}]\ [pos{=}\text{NN}]\ [pos{=}\text{NN}] \end{aligned}$$

We then compute a result set for each strand, using the algorithm from section 5, and finally take the union of the result sets, $ABD \cup ACD$. The query returns 1.6 million results and executes in 1s.

Note that when executing the example above, the subquery AD = [*word*=THE] [] [*pos*=NN] will be used twice. As an optimisation, we cache the results of any duplicated subqueries, to avoid executing them repeatedly.

7.1.1 When to apply disjunction

To expand out disjunctions into strands is not always the most optimal strategy. In particular if the query contains several disjunctions we will get an exponential number of strands.

An alternative strategy is to *not* expand out the disjunction, but rather implement it as set union directly. This means that B and C will be looked up using unary indexes, but we can use the binary index [*word*][][*pos*] to look up AD. Then we can return $AD \cap (B \cup C)$. The problem with this approach is that we would not be able to use the binary indexes [*word*][*pos*] or [*pos*][*pos*].

However, there are even more possibilities. We can also half-expand the disjunction $A(B|C)D$ in two different ways, either into $(AB|AC)D$, or into $A(BD|CD)$. For the first case we can then search the binary index [*word*][*pos*] once for AB and another time for AC, and the unary index [*pos*] for D, and then calculate $(AB \cup AC) \cap D$. And correspondingly for the second case.

So which strategy is the best? It depends on the sizes of the different sets, and we don't know these sizes until we actually calculate them. But a possible heuristic would be to assume that unions are always exclusive, meaning that $|B \cup C| = |B| + |C|$. Using this assumption and the sizes of all the possible seed sets $(A, B, C, D, AB, AC, BD, CD, AD)$ we can calculate which strategy would be the most optimal.

7.1.2 Limitations

The strategy to expand the disjunctions to the top level works for all kinds of disjunctions, but the other strategies may not always work. The semantics described in section 3.3 does not handle all kinds of disjunctions. For query 2(d), we can simply interpret the disjunction as set union:

$$\{\textit{word}^{@0}{=}\text{THE}\} \cap \{\textit{pos}^{@2}{=}\text{NN}\}$$
$$\cap\ (\{\textit{pos}^{@1}{=}\text{JJ}\} \cup \{\textit{pos}^{@1}{=}\text{NN}\})$$

The reason why this works is that the disjuncts have the same length, i.e., that they span the same number of tokens. But when the disjuncts have different lengths, such as in the query

([*pos*=PRON] | [*pos*=DET] [*pos*=NN]) . . .

we cannot know the exact relative position of the token following the disjunction – it will either be 1 (if we matched [*pos*=PRON]) or 2 (if we matched [*pos*=DET] [*pos*=NN]).

In practice, this means that if the disjuncts are of different lengths, and there is a token after the disjunction, then we must expand the disjunction.

For example, suppose that the C subquery of $A(B|C)D$ spans two tokens (e.g., the 2-token query [*pos*=ADV][*pos*=JJ]). Then the query AB will span 2 tokens but AC will span 3 tokens. This means that the final subquery D will have relative position 2 or 3 depending on which disjunct we select. Therefore we cannot calculate $A \cap (B \cup C) \cap D$ or $(AB \cup AC) \cap D$, but are forced to expand the disjunction into two strands ABD and ACD.

Section 8 discusses this case, together with repetition and other regular expression constructs.

7.2 Prefix and suffix queries

Finding all values starting with a given prefix, such as [*word*=CAT .*], is possible using the normal inverted indexes. Since the index is sorted alphabetically, all words matching a given prefix will appear together in the index. Using binary search we can find the start and end positions of all values that match the prefix, but the results will not be one single sorted set. Instead we will get a sequence of sorted groups, one for each matching value, something like $[\underline{12, 43, 57}, \overline{11, 52, 77}, \underline{22, 23}]$. We then have to sort this query set, but this is often quite efficient since the set is already partially sorted.

Unfortunately prefix queries do not play well with binary indexes. Consider the query [THE][CAT .*][RUNS]. We can use a binary index to answer [THE][CAT .*], since all matching bigrams will appear contiguously in the index (*the cat*, *the catcher*, . . .). However, we can not do this for [CAT .*][RUNS], since the matching bigrams may not be contiguous (*cat runs*, *catcher has*, *catcher runs*). Our solution is to ignore binary indexes for token pairs where the first token uses a prefix query.

We implement suffix queries by automatically adding a new annotation to the corpus for each feature, consisting of that feature *backwards*. For example, a token with [*word*=HORSE] is annotated with [*drow*=ESROH] (*drow* is *word* backwards). We transform a suffix query such as [*word*= .* RSE] into the corresponding prefix query [*drow*=ESR .*].

7.3 Regular expressions over values

Consider a query containing a regular expression:

. . . [*word*= .* CAT .+ (ED|ING)] . . .

To execute it, we can exploit the fact that, while the BNC has ≈100 million tokens, it has only ≈1 million *distinct* tokens (the *vocabulary*) – generally the vocabulary of a corpus is much smaller than the corpus as a whole. In our system, the vocabulary is stored alongside the corpus in a plain text file.[6]

First we search the vocabulary file for the regular expression .* CAT .+ (ED|ING). The search returns a list of matching words: *catching*, *scattered*, etc. The regular expression literal is then transformed into a disjunction which is handled as seen earlier:

[*word*=CATCHING] | [*word*=SCATTERED] | . . .

This works well except when the regular expression matches very many words, because our system does not handle the resulting huge disjunction well.

[6]Note that this is not the most space-efficient way of storing a vocabulary – in a production system we would probably use a trie instead (Crochemore and Lecroq, 2020).

8 Future work

Currently our system can only handle a limited number of queries, and there are many more kinds of queries that we want to be able to handle.

8.1 General disjunctions and optional tokens

In section 7.1.2 we already discussed how to handle disjunctions where the disjuncts are of different lengths – and this includes when a token is optional. A simple solution is to expand the disjunctions, but sometimes this might lead to an exponential number of strands. For example, the query $(A|B)(C|D)(E|F)$ contains three disjunctions, but if we expand them we get $ACE|ACF|ADE|\cdots|BDF$ which consists of $2^3 = 8$ strands.

One possible solution could be to let the query sets be sets of *ranges* instead of just positions, where a range is a pair (i, j) of the start and end position of a phrase. Then a query set can contain arbitrary-length phrases. The downside to this solution is that the query sets will become twice as large as before.

8.2 Repeated tokens

Queries with repetitions such as $A\,B^+\,C$, and holes such as A []* C, can perhaps be partially solved using sets of ranges just as for disjunctions.

If we want to make use of binary indexes we can expand a repetition $A\,B^+\,C$ into $AB\,B^*\,C$, which makes it possible to use the binary index AB. Alternatively we can expand in the other direction, into $A\,B^*\,BC$, which makes it possible to use the binary index BC. Which one is the best depends on the sizes of the sets AB, C compared to A, BC, among other things.

Note that we cannot calculate the final query set by taking the intersection of intermediate query sets, because then we would have to keep expanding the repetition indefinitely. Instead we should stop expanding the repetition when we have an intermediate query set of a reasonable size. This intermediate set is guaranteed to contain all matches, but it might contain false positives too. So in the end we have to do a final filtering pass to get only the exact matches, as described in section 8.5.

Holes are a special kind of repetition where we don't know anything about the repeated token, such as in A []* C. For holes it is not useful to expand the repetition, because we still won't be able to make use of any binary index. One possibility is instead to build a tailor-made binary index:

$$[a]\ []^*\ [b] \;\equiv\; \lambda v, w \rightarrow \{a^{@0}{=}v\} \cap \{b^{@k}{=}w \mid k > 0\}$$

"Indexes with holes" can also be used to solve "normal" repetitions: To solve the query $A\,B^+\,C$ we can use the "hole" index [*a*][]*[*b*]. And if we expand the query to $AB\,B^*\,C$ or to $A\,B^*\,BC$, we can also use the binary indexes [*a*][*b*] or [*b*][*c*].

8.3 Regular expressions over tokens

Combinations of sequencing, disjunction, optionality and repetition can be handled using the techniques described above. However, we will quickly get an explosion in the number of ways we can expand queries and decide on the best indexes.

Therefore, to handle general regular expressions over tokens we need to be able to reason about the different expansions and rewrites to come up with an optimal query plan. This is a non-trivial task and something we will look into in the future.

8.4 Regular expressions over values

In section 7.3 we described one way to handle regular expressions over values, such as [*word*= .* CAT .+ (ED|ING)], by searching in the vocabulary and expanding the expression to a long disjunction. However, when there are many possible words matching the regular expression this is not feasible. In those cases we can use an idea from Zobel et al. (1993), where we build an inverted index over character n-grams.

To search for all tokens that match the regular expression above we can search for the ngrams CAT, ED, and ING in this n-gram index, getting the sets A_{CAT}, B_{ED}, and C_{ING}. Now we can compute the new query set $A_{\text{CAT}} \cap (B_{\text{ED}} \cup C_{\text{ING}})$. Note that this result is a query set that might contain false positives, so we will have to filter the final set to get the exact query matches.

8.5 Filtering

The simplest and most general approach is to use filtering. First we translate the query into a less precise query that we can handle, then we filter the results by checking them against the full query. This is how all current corpus engines do, and sometimes this is actually the best approach.

All the techniques we have described in sections 4–7 are the most useful if there is no single search index that returns a reasonable-sized query set. E.g., in the example queries 2(a–b), one of the tokens matches only 33,000 results which is a fairly small set – so it might be the easiest to just filter that set instead of calculating intersections. However, for queries 2(c–d) there are no single small sets so it is much better to use the binary indexes and calculate the intersection. In general our query planner should be able to stop when the query set is small enough, and then resort to filtering instead of continuing with set operations.

8.6 Metadata and multi-layer annotations

The current prototype does not support searching in metadata (such as author, year, language variety, or similar), or multi-layer annotations. This is of course something that must be solved for the system to be useful in practice.

8.7 More efficient set representations

The prototype uses a very simple representation of sets as a sorted array of integers (see section 4). This seems to work well in most cases, but the sets can become quite large. There are several dedicated set data structures that are both compressed and allow for more efficient set operations, such as different kinds of compressed bitmaps (Culpepper and Moffat, 2011; Lemire et al., 2018).

9 Conclusion

We have shown that inverted indexes and efficient set operations can improve searching in large annotated corpora, and in particular binary indexes can improve efficiency by an order of magnitude compared to the traditional unary indexes. By translating queries to set operations, we can use multiple indexes in one query and avoid the need to filter the results afterwards.

We have implemented a prototype which shows promising results, but there is certainly room for improvement. Firstly, the key operations of set intersection, different and union, and building the indexes, can be optimised. Secondly, the query language can be extended to more expressive queries, as discussed in section 8.

It is not always clear how to translate expressive queries to expressions in set theory (see section 7.1.2). An important next step is to find or design a mathematical formalism that queries can be translated into, which is just as amenable to reasoning as set theory is, but supports more expressive queries. We hope that by doing so, we can scale our approach to handle far more complex queries even over huge corpora.

Limitations

The work described in this paper is work in progress. Our results are promising, but we have not extended our approach to more advanced query languages and therefore we cannot be certain how scalable our approach is. Furthermore, we have not done any extensive evaluation and empirical comparison with existing corpus query engines, apart from measuring the runtimes for some example queries, and a limited comparison with Corpus Workbench.

Ethical Considerations

We have not collected any data or made any human experiments when developing the algorithms in this paper, so there are no direct ethical consequences with respect to GDPR or similar. One important consequence of algorithm optimisation is reduced energy consumption, so in the best case this can be a small step in reducing the carbon footprint of digital humanities research.

Acknowledgements

We thank Niklas Deworetzki and the anonymous reviewers for their valuable comments and suggestions.

References

Joachim Bingel and Nils Diewald. 2015. KoralQuery – a general corpus query protocol. In *NODALIDA Workshop on Innovative Corpus Query and Visualization Tools*, pages 1–5, Vilnius, Lithuania.

Maxime Crochemore and Thierry Lecroq. 2020. Trie. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of database systems*, pages 3179–3182. Springer New York.

J. Shane Culpepper and Alistair Moffat. 2011. Efficient set intersection for inverted indexing. *ACM Trans. Inf. Syst.*, 29(1).

Mark Davies. 2005. The advantage of using relational databases for large corpora. *International Journal of Corpus Linguistics*, 10(3):307–334.

Marie-Catherine de Marneffe, Christopher Manning, Joakim Nivre, and Daniel Zeman. 2021. Universal dependencies. *Computational Linguistics*, 47(2):255–308.

Nils Diewald and Eliza Margaretha. 2016. Krill: KorAP search and analysis engine. *Journal for Language Technology and Computational Linguistics*, 31(1):63–80.

Stefan Evert and Andrew Hardie. 2011. Twenty-first century Corpus Workbench: Updating a query architecture for the new millennium. In *Proceedings of the Corpus Linguistics 2011 conference*, University of Birmingham, UK.

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

Peter Kleiweg and Gertjan van Noord. 2020. Alpino-Graph: A graph-based search engine for flexible and efficient treebank search. In *Proceedings of the 19th International Workshop on Treebanks and Linguistic Theories*, pages 151–161, Düsseldorf, Germany.

Thomas Krause and Amir Zeldes. 2016. ANNIS3: A new architecture for generic corpus query and visualization. *Digital Scholarship in the Humanities*, 13(1):118–139.

Daniel Lemire, Owen Kaser, Nathan Kurz, Luca Deri, Chris O'Hara, François Saint-Jacques, and Gregory Ssi-Yan-Kai. 2018. Roaring bitmaps: Implementation of an optimized software library. *Software: Practice and Experience*, 48(4):867–895.

Juhani Luotolahti, Jenna Kanerva, and Filip Ginter. 2017. Dep_search: Efficient search tool for large dependency parsebanks. In *21st Nordic Conference on Computational Linguistics, NoDaLiDa*, pages 255–258, Gothenburg, Sweden.

Udi Manber and Gene Myers. 1993. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5).

Paul Meurer. 2020. Designing efficient algorithms for querying large corpora. *Oslo Studies in Language*, 11(2):283–302.

PostgreSQL. 2024. PostgreSQL documentation. Technical report, The PostgreSQL Global Development Group.

Jonathan Robie, Michael Dyck, and Josh Spiegel. 2017. XML path language (XPath) 3.1. Technical report, W3C.

Jonathan Schaber, Johannes Graën, Daniel McDonald, Igor Mustac, Nikolina Rajovic, Gerold Schneider, and Noah Bubenhofer. 2023. The LiRI corpus platform. In *CLARIN Annual Conference*, pages 145–149, Leuven, Belgium.

Justin Zobel, Alistair Moffat, and Ron Sacks-Davis. 1993. Searching large lexicons for partially specified terms using compressed inverted files. In *19th VLDB Conference*, Dublin, Ireland.