Efficient corpus search using unary and binary indexes

Peter Ljunglöf
Computer Science and Engineering
and Språkbanken Text
University of Gothenburg
Gothenburg, Sweden
peter.ljunglof@gu.se

Nicholas Smallbone
Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden
nicsma@chalmers.se

Abstract

We investigate how disk-based inverted indexes can be used for efficient searching in large annotated corpora. We give a formal semantics for simple corpus queries, and show how they can be translated into lookups in unary and binary indexes.

1 Introduction

The corpus infrastructure *Korp* (Lars Borin, 2012) provides access to more than 250 different corpora comprising a total of 15 billion tokens, via a user-friendly search interface. The main problem with querying Korp is that it is slow – even the simplest searches take a while, and more complex queries can take several minutes.

There are several similar corpus infrastructures around the world, some of them much bigger than Korp, and all struggle with finding new ways to make searching more efficient.

In this paper we give a formal semantics for simple corpus queries, and show how they can be translated into lookups in pre-compiled indexes. We introduce binary indexes which give a drastic improvement for many common queries. We also describe how to implement these indexes in order to optimise intersection of the results.

2 Related work

There seem to be two main approaches to making corpus search efficient: using an inverted index, or an existing database engine.

2.1 Inverted indexes

In the field of information retrieval, the most common inverted index is the *suffix array* (Manber and Myers, 1993). There is plenty of research on how to compress suffix arrays (e.g., Grossi and Vitter, 2005). Meurer (2020) shows how to modify a suffix array to support regular expression search. Suffix arrays can only be used to query plain text: it is not obvious how to use them for querying annotated corpora. Our proposal can be seen as a modified suffix array.

Corpus Workbench (Evert and Hardie, 2011) is the current backbone of Korp. The corpus is compiled into inverted indexes, one per annotated attribute (e.g., word, lemma, part of speech).

2.2 Using a database engine

Davies (2005) translates corpus n-grams into an SQL database, and shows how to use the database to make complex queries. *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, and can compile several different query formalisms into optimised searches.

2.3 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 (or tables) to get a collection of potential search results, and then filter the results one by one, by testing if they match the query. The single indexes (or tables) themselves can be very detailed, but the papers do not discuss how to combine the results of querying different indexes.

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. Each attribute has one single string or numeric value. Our approach is agnostic to which attributes there are.

This definition of corpus is very restricted. We do not currently 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

$$C[0]\ C[1] \ldots C[\mathrm{i}]\ C[i{+}1] \ldots C[n{-}2]\ C[n{-}1],$$

where each token is an attribute-value mapping. We write $C[i].pos$ for the value of attribute *pos* at position i. We also assume that the corpus is divided into sentences, so that certain corpus positions mark the start of a sentence.

3.2 Queries

We use a restricted version of CQL (the "corpus query language", see section 2.2.3 in Evert and Hardie, 2011). The final section discusses how to lift some of the restrictions.

A query is of the form **[***literal****]+**, where a literal is either '*attr* = value' or '*attr* ≠ value'. Here is an example query:

$$[pos\text{=NN}]\ [lemma\text{=to}]\ [lemma\text{=house}\ pos{\neq}\text{VB}]$$

This searches for sentences which contain a noun (NN), followed by the lemma "to", followed by the lemma "house" not as a verb (VB).

3.3 Query semantics

In the query above, the literal "*lemma*=house" occurs 2 tokens after the first query token; we say that it has *relative position* 2. Using relative positions, we can write the example query as:

$$[pos^{@0}\text{=NN}] \wedge [lemma^{@1}\text{=to}]$$
$$\wedge\ [lemma^{@2}\text{=house}] \wedge [pos^{@2}{\neq}\text{VB}]$$

where $[lemma^{@2}\text{=house}]$ means that the lemma at position 2 is house. We 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 $p{+}k$:

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

We call this set a *query set* and we write it $\{attr^{@k}\text{=val}\}$. The semantics of a combined query is then the intersection of the query sets for each of the literals in the query. But if a literal is negated $\{attr^{@k}{\neq}\text{val}\}$, we instead take the difference with the corresponding positive literal. So the semantics of the example query is:

$$\{pos^{@0}\text{=NN}\} \cap \{lemma^{@1}\text{=to}\}$$
$$\cap\ \{lemma^{@2}\text{=house}\} \setminus \{pos^{@2}\text{=VB}\}$$

4 Efficient inverted indexes

Each annotation attribute (*pos*, *lemma*, etc.) can be pre-compiled into an inverted index of corpus positions. This index is inspired by suffix arrays, 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.

Assume that we have a (very small) corpus consisting of the following sentence:

	0	1	2	3	4	5	6
word:	The	horse	raced	past	the	barn	fell
pos:	DT	NN	VB	IN	DT	NN	VB

Now the index for the *pos* attribute will be the following array of positions: [0, 4, 3, 1, 5, 2, 6]. This array is sorted alphabetically on *pos* values: [0, 4] are DT, [3] is IN, [1, 5] are NN, and [2, 6] are VB. Furthermore, each group of positions for the same value is in increasing order.

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 in the example index these searches return 3 and 4, which are the start and end indices in the index, where all the corpus positions for NN are found.

Naively, to execute the query $\{pos^{@k}\text{=NN}\}$, we should search for NN, and then subtract k from all matching positions. For efficiency, we instead just record the start and end indices (3 and 4) and the relative position k, using which we can easily recover all matching positions.

4.2 Intersecting query sets

Since the corpus positions within a query set are ordered, we can efficiently intersect two query sets. Depending on the relative sizes of the sets we use one of the following two algorithms:

- If one set is much larger than the other, we 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.

- If the sets are approximately the same size, we use a merge strategy: Iterate through both sets in parallel.

Regardless of the intersection algorithm, the resulting set will also be ordered, so we can use that when intersecting with more query sets.

We adapt the algorithms to compute set difference, which is used for negated literals. We also adapt them to work on query sets that include a relative position k, as in section 4.1.

4.3 Indexes and sets as on-disk arrays

The query indexes and the query sets are stored as memory-mapped integer arrays on disk. In this way we can work with huge corpora that are much too large to fit in RAM.

4.4 Computing the query result

To execute a query, we look up each literal (section 4.1) to get its query set. Then we follow the query semantics, using set intersection and difference (section 4.2) to find the final result. If we have several query sets, we have to decide in which order to intersect them. A heuristic that works well in practice is to start from the smallest sets and leave the largest until later.

We did some testing with the 100 million token British National Corpus (BNC).[1] The resulting query sets for the example query from section 3.2–3.3 are as follows:

$\{pos^{@0}\text{=NN}\}$	→	26 M results
$\{lemma^{@1}\text{=to}\}$	→	2.6 M results
$\{lemma^{@2}\text{=house}\}$	→	63 k results
$\{pos^{@2}\text{=VB}\}$	→	18 M results

We start by intersecting $\{lemma^{@2}\text{=house}\}$ with $\{lemma^{@1}\text{=to}\}$, which gives 586 results. Then we intersect with other query sets, in the end finding 62 search results. Intersection here uses the *filtering strategy* from section 4.2, which only needs to iterate through the smallest index, $\{lemma^{@2}\text{=house}\}$, so the query runs quickly.

5 Binary query indexes

Consider the following very general query:

$$[lemma^{@0}\text{=the}] \wedge [pos^{@1}\text{=NN}] \wedge [pos^{@2}\text{=NN}]$$

Each of these literals results in a huge set:

$\{lemma^{@0}\text{=the}\}$	→	6 M results
$\{pos^{@1}\text{=NN}\}$	→	26 M results
$\{pos^{@2}\text{=NN}\}$	→	26 M results

So the intersection becomes slower (about 20 times slower than the previous example). The first intersection gives 4.2 M results, and the second one results in 570,000 final results.

To improve the efficiency of these generic queries we can also build *binary* query indexes, in addition to the unary indexes.

5.1 Binary indexes

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

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

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

$$[a]\,[b] \;\equiv\; \lambda\text{v, w} \rightarrow \{a^{@0}\text{=v}\} \cap \{b^{@1}\text{=w}\}$$
$$[a]\,[]\,[b] \;\equiv\; \lambda\text{v, w} \rightarrow \{a^{@0}\text{=v}\} \cap \{b^{@2}\text{=w}\}$$
(similar for $[a]\,[]\,[]\,[b]$, etc.)

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

5.2 Searching using binary indexes

Now, if we take the example from the beginning of section 5, we can search in the following binary indexes, in addition to the unary indexes we already tried:

[*lemma*=the] [*pos*=NN]	→	4.2 M results
[*lemma*=the] [] [*pos*=NN]	→	1.9 M results
[*pos*=NN] [*pos*=NN]	→	3.7 M results

Now we can intersect the two smaller sets:

$$\{lemma^{@0}\text{=the}\} \cap \{pos^{@2}\text{=NN}\}$$
and
$$\{pos^{@1}\text{=NN}\} \cap \{pos^{@2}\text{=NN}\}$$

This intersection gives 570 k 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.

5.3 Reducing the size of binary indexes

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.

We don't have to build all these indexes. If a binary index is missing, we can simply fall back to searching in two unary indexes instead, just as in section 4. But the binary indexes are very useful, so can we reduce their size in any way?

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

We observe that if a query uses a literal that is uncommon in the corpus (e.g. *lemma*=turtle), there is no need to use binary indexes for that query, since the unary *lemma* index will already give us a small query set. So one optimisation is to not add all value pairs (v, w) to the index, but only the ones where v and w are common:

- 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 set the threshold to 10,000 unary instances, the binary indexes are reduced to 250–300 MB each.

6 Sentence borders

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 our query from section 5 to match a sentence that ends in "the horse" where the next sentence starts with "cats".

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 S) only for the tokens that start a sentence. Our example query can then be translated to:

$$[\mathit{lemma}^{@0}\text{=the}] \land [s^{@1}{\neq}\text{S}] \land [\mathit{pos}^{@1}\text{=NN}] \\ \land [s^{@2}{\neq}\text{S}] \land [\mathit{pos}^{@2}\text{=NN}]$$

6.1 Sentence borders and binary indexes

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

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

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 in section 5 can still be translated to searches in the following three binary indexes:

[*lemma*][*pos*], [*lemma*][][*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 Future work

We have a prototype written in Python. It is fast, despite Python being an interpreted language, but there is certainly room for improvement. In particular building the indexes can be optimised.

7.1 More expressive queries

Currently we can only handle simple queries with exact matching and conjunction (i.e., set intersection), but there are some possibilities for making it more expressive:

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.

Disjunction Disjunctive queries such as ([*pos*=DT] | [*pos*=JJ]) [*pos*=NN] should be fairly easy to support, by using set disjunction in a similar way to intersection.

Repetition Queries with repetitions such as [*pos*=DT] [*pos*=JJ]* [*lemma*=house], and holes such as [*pos*=DT] []* [*lemma*=house], can probably be solved by building tailor-made binary indexes.

Prefixes Finding all values starting with a prefix, such as [*lemma*=cat...], is more tricky but still possible to solve. Binary search 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 [12 43 57] [11 52 77] [22 23]. We then have to do a second pass to merge all these groups into one single set.

Regular expressions, backreferences, etc., etc. Hopefully we can incorporate ideas from the literature, such as the techniques for regular expression matching by Meurer (2020), or the graph search used in *AlpinoGraph* (Kleiweg and van Noord, 2020), or the query translations of *Krill* (Diewald and Margaretha, 2016), just to mention a few.

References

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

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.

Roberto Grossi and Jeffrey Scott Vitter. 2005. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2).

Peter Kleiweg and Gertjan van Noord. 2020. Alpinograph: 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.

Johan Roxendal Lars Borin, Markus Forsberg. 2012. Korp – the corpus infrastructure of Språkbanken. In *Proceedings of LREC 2012*, pages 474–478, Istanbul, Turkey.

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. In Hagen, Hjelde, Sternholm, and Vangsnes, editors, *Bauta: Janne Bondi Johannessen in memoriam*, Oslo Studies in Language, 11(2), pages 283–302.