Querying Large Treebanks: Benchmarking GrETEL Indexing

Bram Vanroy* BRAM.VANROY@UGENT.BE
Vincent Vandeghinste** VINCENT.VANDEGHINSTE@KULEUVEN.BE
Liesbeth Augustinus** LIESBETH.AUGUSTINUS@KULEUVEN.BE

**Language and Translation Technology Team (LT^3), Ghent University, Belgium*

***Centre for Computational Linguistics, KU Leuven, Belgium*

Abstract

The amount of data that is available for research grows rapidly, yet technology to efficiently interpret and excavate these data lags behind. For instance, when using large treebanks for linguistic research, the speed of a query leaves much to be desired. GrETEL Indexing, or GrInding, tackles this issue. The idea behind GrInding is to make the search space as small as possible before actually starting the treebank search, by pre-processing the treebank at hand. We recursively divide the treebank into smaller parts, called subtree-banks, which are then converted into database files. All subtree-banks are organized according to their linguistic dependency pattern, and labeled as such. Additionally, general patterns are linked to more specific ones. By doing so, we create millions of databases, and given a linguistic structure we know in which databases that structure can occur, leading up to a significant efficiency boost. We present the results of a benchmark experiment, testing the effect of the GrInding procedure on the SoNaR-500 treebank.

1. Introducing GrETEL

GrETEL (**Gr**eedy **E**xtraction of **T**rees for **E**mpirical **L**inguistics) is a linguistic search engine that makes it possible to query syntactically annotated corpora or *treebanks* in a user-friendly way.[1] Linguists often make use of corpora to obtain empirical evidence or to gather quantitative information for their research. While treebanks are of special interest for syntactic research, some linguists are a bit deterred to use them because of their high threshold of exploitation, due to the limited user-friendliness of the search tools and the lack of standardization in grammatical framework, formalism, and query languages. GrETEL aims to overcome those issues.

GrETEL is an online system, which has the advantage that it is platform-independent and does not require any local installation of treebanks nor parsers. Other online treebanking platforms that share these advantages include INESS-Search (Meurer 2012), TüNDRA (Martens 2013), and PaQu (Odijk 2015). The key difference between GrETEL and those other search tools is that GrETEL offers *example-based* querying. Linguists tend to use example sentences as a starting point for their research. In GrETEL, they can use those examples as input to begin the treebank search.

GrETEL offers two modes to query a treebank: via a natural language example or via an XPath query.[2] Its architecture is illustrated in Figure 1.

The first mode to query a treebank is the example-based querying method. First, the user provides an example construction containing the phenomenon under investigation. Second, the input is syntactically analyzed with the Alpino parser (van Noord 2006). Third, the user indicates which parts of the example sentence are important for the treebank search and which information of each part has to be included (lemma, POS, token). This information is automatically converted into an XPath expression, which can be used to query the treebank. The generated XPath expression may be further adapted by the user. Fourth, the user selects the treebank(s) that should be queried.

1. The latest version of the tool is available via `http://gretel.ccl.kuleuven.be`.
2. `https://www.w3.org/TR/xpath`

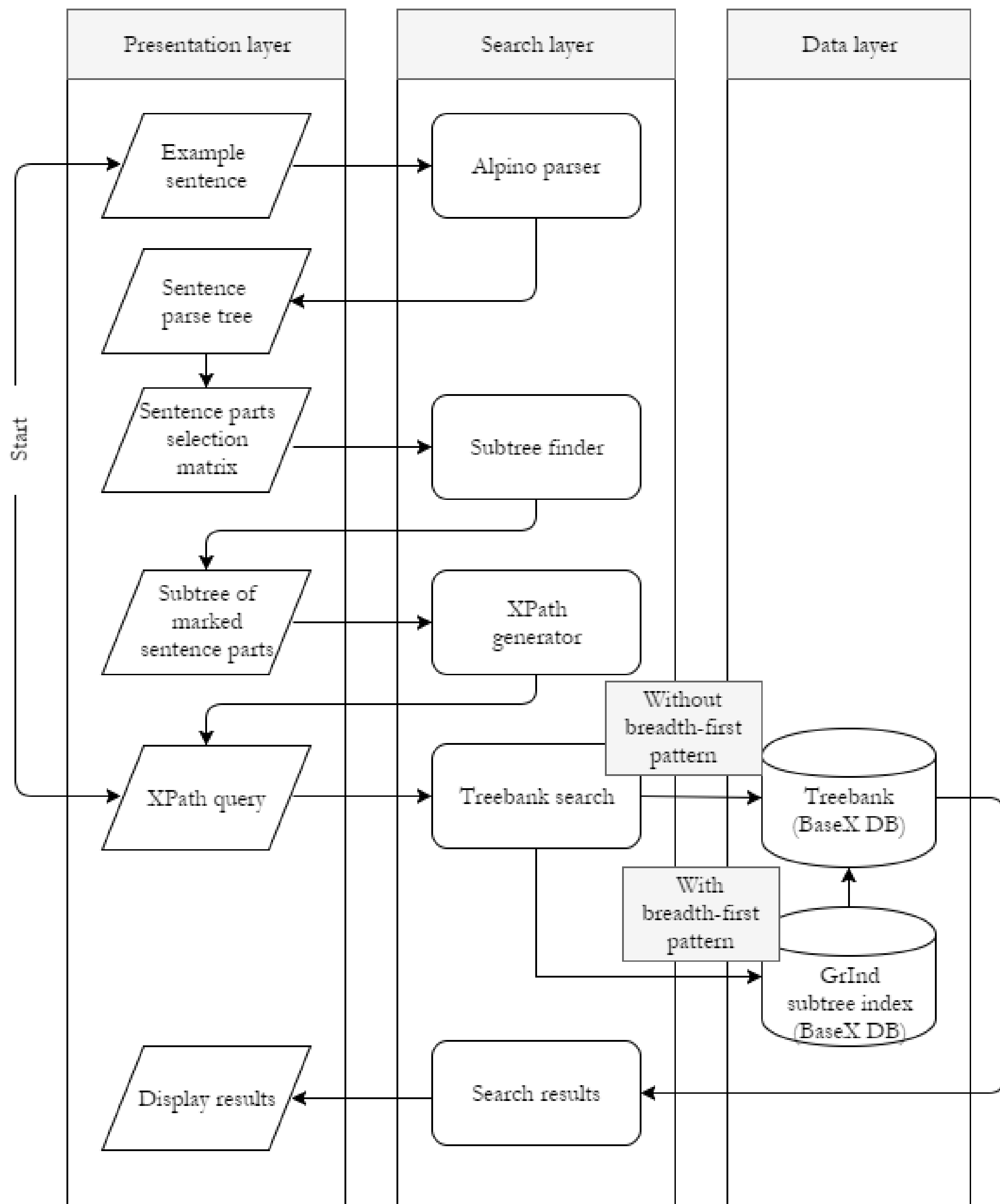


Figure 1: Architecture of GrETEL

Fifth, a query overview is given. In the sixth and final step, the actual treebank search is performed and the results, if any, are presented to the user.

The second method to query a treebank with GrETEL is by entering an XPath query straight away. This is a more flexible query method, but it requires that the user masters this formal query language. Because our pre-processing step creates millions of databases, as will be discussed in

section 3, we chose BaseX[3] as our database system. BaseX is an XML database that scales very well, which means that it is able to deal with large corpora or large quantities of XML files. For a more detailed description of the GrETEL work flow, cf. the examples in Augustinus et al. (2012) and Augustinus et al. (in press).

2. Querying large treebanks

The first and second version of GrETEL enabled users to query the CGN Treebank (van der Wouden et al. 2002) for spoken Dutch and the LASSY Small treebank (van Noord et al. 2013) for written Dutch. Those treebanks each contain circa one million words. The annotations of these corpora are manually verified, which makes the treebanks a valuable resource for linguistic research. The relatively small size of the treebanks is a downside, however; if one is looking for rare phenomena, only a handful or no results can be found. The construction of (very) large, automatically created treebanks offers a solution to this problem. The SoNaR treebank was created during the Lassy project (van Noord et al. 2013), and is part of the Lassy Large treebank.[4] It consists of the SoNaR corpus (Oostdijk et al. 2013) (500 million words of Dutch text), parsed with the Alpino parser.

There are several tools and databases available to query *flat* (raw or POS-tagged) large corpora, such as the OpenSonar project (Reynaert et al. 2014), some of which scale up well even if billions of words are included in the data, see for instance the English corpora available via `http://corpus.byu.edu` . In contrast, making large treebanks searchable online in *reasonable* time is still a nontrivial task, at least if one allows querying for entire tree structures, rather than doing a string search or querying basic head-dependent relations. At our first attempt towards including the SoNaR treebank in GrETEL we bumped into scalability issues, especially with respect to query times. As a solution, we came up with GrInding, an indexing system for treebanks.

3. The concept of GrInding

We scale up syntactic search from a medium size treebank (circa one million words) to a very large treebank (circa 500 million words), using a method called *Gr*ETEL *Ind*exing or *GrInding* (Vandeghinste and Augustinus 2014). Here, we describe the updated version of GrInd. Conceptually it is similar to the method described in Vandeghinste and Augustinus (2014) with some modifications such as an added parameter (cf. Item 1), and some bug fixes.

The general idea behind this approach is to make the search space as small as possible before starting the actual treebank search. In other words, if you know under which tree to find the pot of gold, you do not need to cut down the whole forest. We transpose this to a scenario where the pot of gold is the collection of XML structures matching the input query by organizing (sub)trees according to the syntactic pattern they contain. If you know in which database the results are stored by analyzing the input query, you do not have to look in the irrelevant databases, which saves a lot of search time.

The XML representation of each sentence in the treebank is recursively divided into its bottom-up subtrees, i.e. all subtrees that have lexical elements in their terminal nodes. We organize the data into databases that contain all bottom-up subtrees for which the two top levels (i.e. the root and its children) adhere to the same syntactic pattern. As an example, consider the parse tree in Figure 2. In a top-down fashion, we take for each node all the bottom-up subtrees. For instance, the subtree generation for the node `--|smain` is illustrated in Figure 3. GrInding the `--|smain` node results in seven subtrees: one with three children (Figure 3a), three with two children (Figure 3b-d), and three with one child node (Figure 3e-g). This procedure is applied recursively for every node in the parse

3. `http://basex.org`
4. According to `https://www.let.rug.nl/vannoord/Lassy/` Lassy Large is a 700 million word corpus with automatically assigned syntactic annotations. Lassy Large contains, amongst other corpora, the SONAR500 corpus, consisting of 41 million sentences, 510 million tokens.

tree. To avoid unnecessary computations we have added some parameters to the GrInd process to filter out subtrees that have dimensions that exceed our needs, or that are simply unrealistic, indicating a faulty parse attempt. The parameters are:

(1) a. `$maxWidth=20` is the maximum amount of daughters a root node can govern. It is currently set to 20 as we believe it is unlikely that GrETEL users will look for subtrees with a root node that has more than 20 children.

b. `$maxChildrenSubtree=7` sets a ceiling for the length of the breadth-first pattern. Whereas `$maxWidth` limits the size of the (sub)tree, `$maxChildrenSubtree` puts a restraint on the size of the generated patterns for that tree. In practice this means that a tree with 20 daughters is included in the searchable data, but you can only query it with a breadth-first pattern of seven nodes at most. This parameter has only recently been added. Previous versions of GrInd as discussed in Vandeghinste and Augustinus (2014) did not have this constraint. By adding the parameter we decrease the processing needed (as the possible combinations of the subtree's children are greatly reduced) without having a large effect on the querying options: as long as a user does not exceed a query that implies a breadth-first pattern with more than seven children, the subtree can be found. Note that this does not mean that the input XPath can only contain seven nodes in total. It means that the XPath code can only contain seven nodes at the second level.

c. `$maxDescendants=1000` is limiting the number of descendants (nodes) in a tree, not GrInding the tree if it has more than n descendants. This variable is a safety net; `$maxWidth` and `$maxChildrenSubtree` already put restrictions on the outcome of GrInding but `$maxDescendants` makes sure that extreme outliers (most probably parsing errors) are excluded as well.

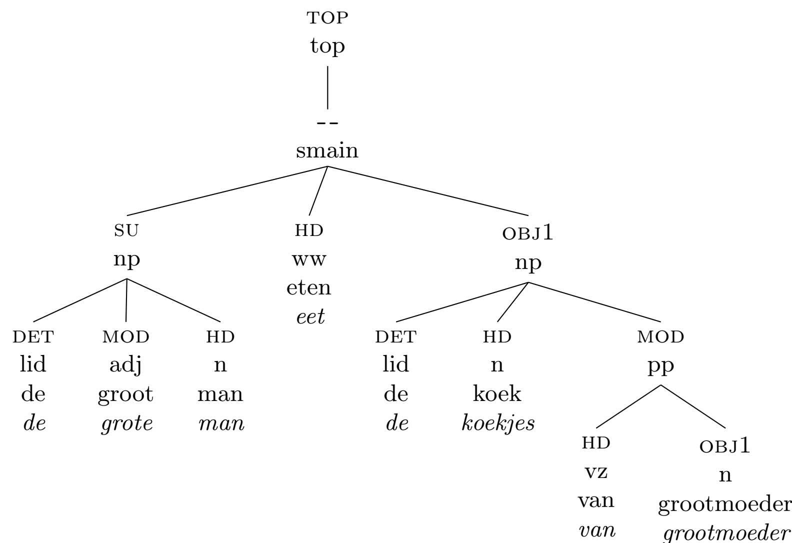


Figure 2: Parse tree of the sentence *De grote man eet de koekjes van grootmoeder* 'The tall man eats grandmother's biscuits'

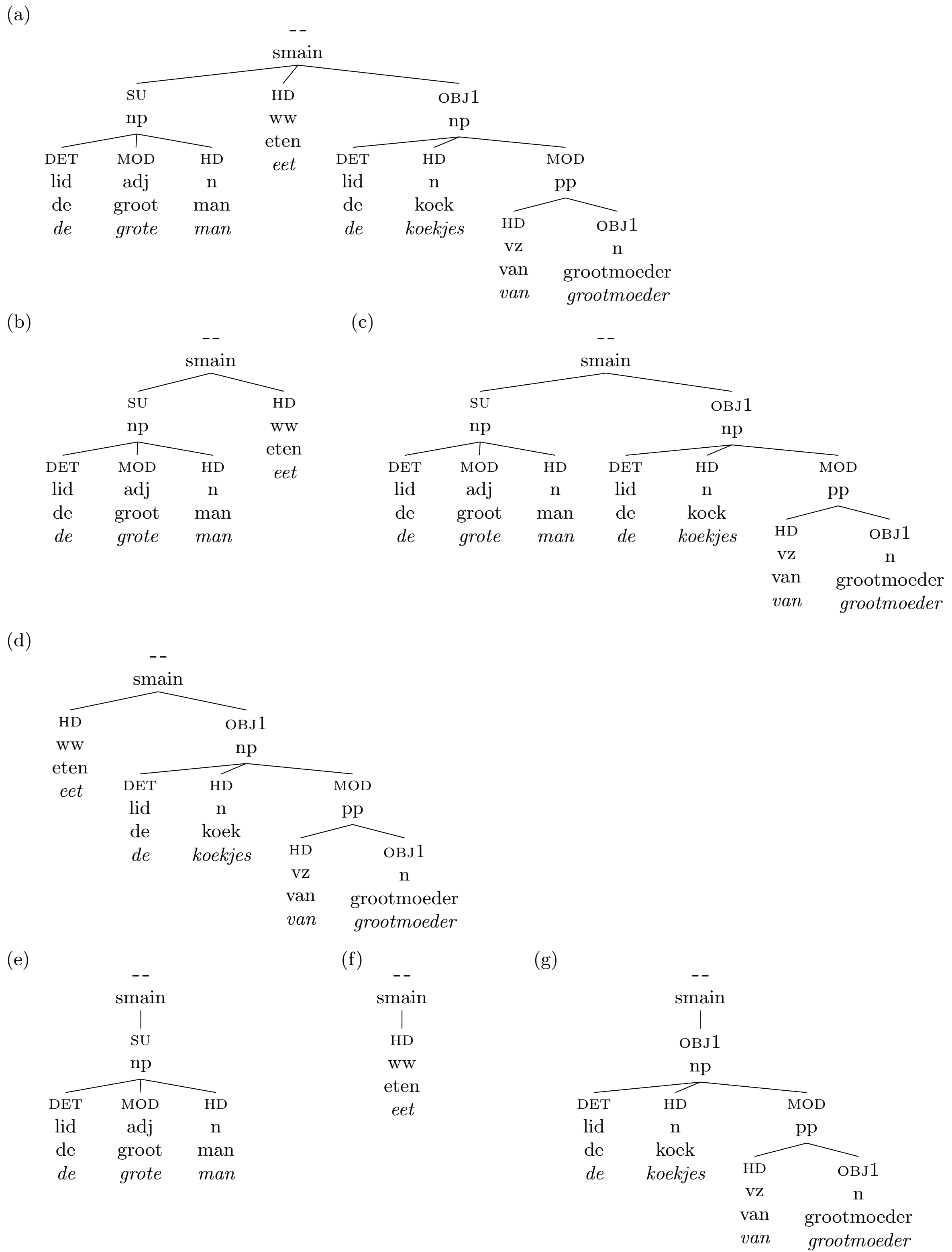


Figure 3: Subtree generation for the node subtree generation for the node `--|smain` of the parse tree in Figure 2

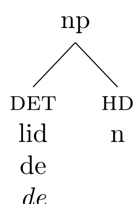
For every extracted subtree, we convert the children of the root into a string-based breadth-first pattern, after Vandeghinste and Martens (2010). The pattern of each subtree is determined by its two topmost levels, i.e. the top and its child nodes. For each of the child nodes the dependency relation and the syntactic category (for non-terminals) or the POS tag (for terminal nodes) are joined by a `%`-sign, e.g. `su%np, hd%ww` for the subtree in Figure 3b. The strings of all child nodes are then sorted alphabetically and concatenated with an underscore separator (e.g. `hd%ww_su%np`). Finally, the topmost node's category is prepended to the whole string (e.g. `smainhd%ww_su%np`). Next, all subtrees are organized according to their syntactic breadth-first pattern per corpus component. All subtrees matching the same pattern are placed into the same file, adding the sentence identifiers indicating where these subtrees come from. Each file is then put into a separate database.

Each subtree has some syntactic (breadth-first) pattern A, B, ... Z. By grouping all subtrees with an identical pattern, we can create databases A, B, ... Z that only contain subtrees with the patterns A, B, ... Z respectively. If a user then searches for an input example that agrees with pattern B, the database software can immediately start looking in database B without having to look in A, or in any other database. This way, the search space is immediately cut down to a manageable size.

As SoNaR is a huge treebank it is not surprising that it contains a massive amount of different syntactic patterns. In total, more than 17 million databases are created.

We implemented a mechanism to avoid an even bigger explosion of the data, i.e. the concept of *includes*. We explain this idea by giving an example. The subtree in 2a is a tree representation of an input query where a user looks for a noun phrase consisting of a determiner that has to be the Dutch word *de* 'the' and any noun. This tree can be generalized as the breadth-first pattern in 2b and can be found in the treebank by means of the XPath code in 2c. In other words, the breadth-first pattern is not identical to the XPath structure, but merely a generalization of it. Noun phrases with a determiner, any noun, and any modifying prepositional phrase such as the subtree in (3a) (corresponding to the pattern in (3b)) should also be found by the XPath structure in (2c), because the former also contains a noun phrase with *de* 'the' and any noun, and adds a prepositional phrase. In other words, subtree (3a) extends subtree (2a). More specific patterns are included in more general patterns, so the noun phrases containing a determiner, a noun, and prepositional phrase should be included in the database of NPs with only a determiner and a noun. In order to avoid duplication of the data this is done by placing a reference to the database of the larger pattern (the *include*) in the database of the smaller pattern, rather than copying all the patterns.

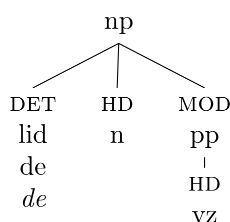
(2) a.



b.
```
npdet%lid_hd%n
```

c.
```
//node[@cat="np" and node[@rel="det" and @pt="lid" and
    @word="de" and @lemma="de"] and node[@rel="hd" and @pt="n"]]
```

(3) a.



b. npdet%lid_hd%n_mod%pp

The code example in Figure 4 shows part of the database file for `npdet%lid_hd%n` in the treebank component WR-P-E-E.

```
<tree id="WR-P-E-E-0000000018.p.2.s.1">
  <node begin="13" cat="np" end="15" id="24" position="0" rel="su">
    <node begin="13" end="14" frame="determiner(de)" id="25" infl="de"
        lcat="detp" lemma="de" lwtype="bep" naamval="stan" npagr="rest"
        pos="det" postag="LID(bep,stan,rest)" pt="lid" rel="det"
        root="de" sense="de" word="de"/>
    <node begin="14" end="15" frame="noun(de,count,pl)" gen="de"
        getal="mv" graad="basis" id="26" lcat="np"
        lemma="vertaalcomputer" ntype="soort" num="pl" pos="noun"
        postag="N(soort,mv,basis)" pt="n" rel="hd"
        root="vertaal_computer" sense="vertaal_computer"
        word="vertaalcomputers"/>
  </node>
</tree>
<include file="npdet%lid_hd%n_mod%pp"/>
```

Figure 4: XML containing subtrees from the WR-P-P-E `npdet%lid_hd%n` database

If a user looks for an input example which is transformed from the subtree (2a) into the XPath code in (2c), GrETEL converts this structure into the breadth-first pattern (2b). BaseX opens the database corresponding to this pattern and looks for all subtrees that agree with the XPath code.

After finding all possible subtrees that match the pattern in (2a) specifically, an `<include>` tag is found which indicates that another database exists with an extended pattern, which also matches the query in (2c). BaseX is then instructed to keep digging in the database(s) of the corresponding include(s). First it looks for matching subtrees, and then it checks whether there are any additional includes in that database. This process is repeated recursively until no more includes are found. Subtrees that are found in the database matching pattern (3b) also match the pattern in (2b), but subtrees residing in database (2b) do not agree with the (extended) pattern in (3b): an NP consisting of a determiner and a noun is contained in an NP with a determiner, a noun, and a PP as children, but not vice versa.

When the complete treebank has been GrInded, the created files still need to be *regularised*. The new files are not valid XML yet because we have always appended new subtrees or includes, but there is no root element. For each GrInd file, a new root `treebank` is created, which consists of two daughter nodes: `trees` and `includes`. The generated subtrees and includes are then put in their respective parent node. As a result, the GrInd files are turned into valid XML.

As a consequence of how a GrInded corpus is built, it is not possible to add newly GrInded trees to an already GrInded corpus or update existing trees. Because the GrInd files contain all subtrees of a specific pattern, it is not straightforward to update a single subtree in that file from a file manipulation perspective. Also, if a tree changes and a breadth-first pattern does not exist anymore in one of its subtrees, it is not easy to find the file that contains the now obsolete subtree and remove subtree from it. Furthermore, trying to update the GrInd files would also mean that they need to be regularised again, or de-regularised and regularised again. Even if updating the GrInd files themselves would work, it still would not be a trivial task to update the created BaseX databases. If the complete treebank should not be imported completely from the start to save time,

it should be known which files have been adapted as well which files can be deleted. Only then a successful update to BaseX is possible. All of this to say, that updating an existing, GrInded treebank is not trouble free.

The GrInding process can be summed up as follows: first each individual XML tree (corresponding with a sentence) is recursively divided into bottom-up subtrees. To each subtree we then add a reference (`id` attribute) to the sentence it belongs to so the entire sentence can be retrieved (cf. Figure 4). These subtrees are then converted into a breadth-first pattern. Next, each subtree is assigned to an XML file which contains all subtrees that adhere to the same breath-first pattern. Finally, if one pattern is actually an extended version of another, a reference to the database of the larger, more specific pattern is included in the smaller one.

4. Benchmarking GrInd

In theory the GrInding process is an effective way to reduce the search space. However, having millions of databases also requires the database system to make a tremendous amount of sequential connections which might have a negative influence on the query time. In order to measure the impact of the GrInding process (positive or negative), a benchmark was set up. As described in section 4.1, 97 queries were selected to compare the querying speed of the baseline version and the GrInded version of the SoNaR treebank. By selecting a variety of XPath structures it is possible to investigate on which type of queries the GrInding process has the largest or the smallest impact.

In section 3 the GrInding process was explained. It results in millions of databases corresponding to the syntactic breath-first patterns of the bottom-up subtrees. The baseline version of SoNaR has not been GrInded but it does not exist of a single database either. Each component is split up into files, so-called treebank parts, of about 10,000 sentences each.[5] This implies that the baseline version also has to make multiple database connections. To give an idea of the scale: the largest component of SoNaR (WR-P-P-G) contains 14,974,007 sentences, which comes down to 1506 databases (treebank parts) in the baseline version whereas the GrInded version is divided over 4,032,557 databases (breadth-first patterns). We could have merged all treebank parts for each component into one large XML file per component to present the baseline. However, we chose to use the data as provided to us to show the difference in performance in a real-world application. Table 1 provides an overview of the number of baseline databases, GrInded databases, sentences and tokens per treebank component. We will come back to the description of the SoNaR-corpus in subsection 4.3.

In this section we will discuss how we created a test set of XPath codes (4.1), followed by a number of points of interest that needed to be considered when writing the benchmark script (4.2). Next we will go over the set-up of the benchmark test (4.3).

4.1 Creating a test set

In order to build a test set for the benchmarking experiment, we use the logs of GrETEL as a starting point. GrETEL's input is (anonymously) logged since November 2014. The analysis of the XPath queries from the logs shows that about 95% of the queries include patterns that are no more than six levels deep, nor contain more than 14 nodes in total. We have used this range as a representative restriction on the test set, so only patterns within this range are included; we want to see how well GrInd performs on our systems with the most frequent queries. However, this does not imply that GrInd only works on this restricted set of XPath queries. In section 4.2 we discuss the restrictions for the XPath that can be efficiently queried in a GrInded corpus. Below, we give an example of an XPath pattern of depth six (4a), and an example with 14 nodes (5a).

The latter query contains an internal structure that is not present in the former: `number(@begin) < number(X)`. This piece of code ensures that the value of the `begin` attribute of the current node is

5. This is how the treebank is delivered to us by the TST-centrale, and is probably the size of the batches that were sent to the high performance cluster for parallel parsing these large volumes.

component	contents	baseline dbs	grind dbs	sentences	tokens
WR-P-E-A	Discussion lists	441	2,831,430	4,396,361	56,973,211
WR-P-E-C	E-magazines	56	637,565	551,343	8,625,948
WR-P-E-E	Newsletters	1	2 096	115	1917
WR-P-E-F	Press releases	2	638,363	18,373	332,766
WR-P-E-G	Subtitles	409	651,365	3,925,834	28,209,131
WR-P-E-H	Teletext pages	5	75,938	40,715	448,865
WR-P-E-I	Websites	23	233,225	205,037	3,111,568
WR-P-E-J	Wikipedia	136	1,002,855	1,355,061	22,976,434
WR-P-E-K	Blogs	1	54,225	8616	139,765
WR-P-E-L	*Tweets*	333	–	2,636,866	23,197,200
WR-P-P-B	Books	209	1,434,868	1,710,131	26,184,247
WR-P-P-C	Brochures	10	172,004	74,921	1,213,301
WR-P-P-D	Newsletters	1	19,817	2185	33,529
WR-P-P-E	Guides & manuals	3	41,166	19,077	236,070
WR-P-P-F	Legal texts	67	391,066	650,509	10,689,681
WR-P-P-G	Newspapers	1506	4,032,557	14,974,007	211,659,990
WR-P-P-H	Periodicals & magazines	551	2,733,406	5,476,086	93,023,716
WR-P-P-I	Policy documents	44	698,337	387,534	8,711,354
WR-P-P-J	Proceedings	2	73,244	16,938	314,025
WR-P-P-K	Reports	11	354,737	93,565	2,218,223
WR-U-E-A	*Chats*	188	–	2,387,147	11,873,184
WR-U-E-D	*SMS*	13	–	101,116	723,876
WR-U-E-E	Written assignments	3	80,059	23,498	357,947
WS-U-E-A	Auto cues	217	894,465	2,163,661	28,086,331
WS-U-T-B	Texts for the visually impaired	5	163,762	44,866	674,966
TOTAL		4237	17,216,550	41,263,562	540,017,245

Table 1: Overview of the amount of databases in the SoNaR-500 treebank

smaller than the `begin` value of node `X`. In other words, the pattern has a fixed order where the current node needs to be in front of the node specified by `X`. The option to search for a fixed-order construction is available in the example-based search mode of GrETEL. Queries without this order constraint, such as (4a), do not take into account the order of the nodes. For both XPath constructions, the breadth-first pattern is given as an additional example of the process.

(4) a.
```
//node[@cat="smain" and node[@rel="mod" and @cat="pp" and
    node[@rel="hd" and @pt="vz"] and node[@rel="obj1" and
    @cat="np" and node[@rel="det" and @pt="lid"] and
    node[@rel="hd" and @pt="n"]]] and node[@rel="hd" and
    @pt="ww"] and node[@rel="su" and @pt="vnw"] and
    node[@rel="obj2" and @cat="np" and node[@rel="det" and
    @pt="lid"] and node[@rel="hd" and @pt="n"]] and
    node[@rel="obj1" and @cat="np" and node[@rel="det" and
    @pt="lid"] and node[@rel="hd" and @pt="n"]]]
```

b. smainhd%ww_mod%pp_obj1%np_obj2%np_su%vnw

(5) a.
```
//node[@cat="pp" and node[@rel="hd" and @pt="vz" and
    number(@begin) < number(../node[@rel="obj1" and
    @cat="np"]/node[@rel="mod" and @cat="pp"]/node[@rel="hd"
    and @pt="vz"]/@begin)] and node[@rel="obj1" and @cat="np"
    and node[@rel="mod" and @cat="pp" and node[@rel="hd" and
    @pt="vz" and number(@begin) < number(../node[@rel="obj1"
    and @cat="np"]/node[@rel="mod" and
    @cat="pp"]/node[@rel="hd" and @pt="vz"]/@begin)] and
    node[@rel="obj1" and @cat="np" and node[@rel="mod" and
    @cat="pp" and node[@rel="hd" and @pt="vz"]]]]]]
```

b. pphd%vz_obj1%np

To search in the GrInded version of the corpus all XPath test cases have to be transformed into breadth-first patterns as well. As explained in section 3, that pattern defines in which database the BaseX server should look in order to find matching tree structures. The conversion from XPath to breadth-first was done using a Perl script.[6] We manually verified the generated patterns. As a result, all original XPath structures are now accompanied by their breadth-first counterpart.

The resulting test set consists of 97 XPath expressions and their corresponding breadth-first patterns, which will be limited to 87 as discussed in section 4.2 below.

4.2 A priori considerations on the benchmark

The benchmark of each SoNaR component was done on the same machine (section 4.3), and the benchmark scripts for each variant of the corpus (GrInd and baseline) are as similar as possible in order to make a fair comparison between the two approaches.

To formalize a caching effect, if any, a benchmark was conducted on a single component (WR-P-E-C), and each query was run multiple times subsequently. A caching effect is not especially relevant because it is the very first look-up that is important in a real-world scenario, but nonetheless a caching effect is interesting to take a look at. Caching will be discussed together with the other results of WR-P-E-C in section 5.2.

After running some initial tests on a small scale, it became clear that 10 out of the 97 cases are not worth further investigation. It concerns extreme cases where the breadth-first pattern is as underspecified as possible. In section 3 we already explained how the breadth-first pattern of a subtree is formed: the parent node's category, followed by each child node's `rel` attribute, a %-sign, and its `cat` or `pt` attribute. The patterns of the child nodes are ordered alphabetically and separated by an underscore (`_`). The GrETEL interface provides an option to ignore the top category of the query tree. This is useful if users want to look for general patterns, e.g. when they want to search for a pattern that may occur in subordinate clauses as well as in main clauses. In a simple noun phrase, this would look like the XPath code in (6a). This XPath expression is then used to generate a breadth-first pattern. When no `@cat`-value is specified for the topmost node, the string `ALL` is used at the start of the breadth-first pattern instead. For the XPath code in 6a the pattern would look like (6b).

(6) a.
```
//node[@cat and node[@rel="det" and @pt="lid"] and
    node[@rel="hd" and @pt="n"]]
```

6. In the GrETEL interface, this is done behind the scenes by a PHP function.

b. ALLdet%lid_hd%n

A breadth-first pattern without a specific top category is a very general construction. If such a pattern is sent to the server, `ALL` is replaced by all 26 possible syntactic categories. This obviously results in more queries than when a category had been specified. This is not a problem as such as the pattern of the child nodes (`det%lid_hd%n`) is specific enough, and the search space is still greatly reduced compared to the baseline. Things get out of hand when the children are underspecified as well. An example of such a case is when a user only gives one word as an input. An example XPath for such a construction is given in (7).

(7)
```
//node[@pt="vnw" and @status="vol" and @genus="masc" and
    @vwtype="pers" and @getal="ev" and @persoon="3" and
    @naamval="nomin" and @pdtype="pron"]
```

As the query in (7) does not contain a `cat` nor a `rel` attribute, no breadth-first pattern is generated. The GrInd script does not take into account constructions in which only a lexical element (containing a `pt` attribute) is defined. We consider queries such as (7) edge cases, as GrETEL is intended to search for dependencies, and not for single words or strings of words that have no defined relationship between them. It expects a `rel` attribute indicating the relationship between two or more nodes. Because no syntactic pattern can be generated in these cases, the string `ALL` is used instead as a breadth-first pattern. The same principle as with the pattern in (6b) is applied here: `ALL` is replaced by every possible category. In practice this means that the entire corpus is queried, as each category and all its sub-patterns are queried, thus nullifying the goal of GrInd. Because of this, the ten XPath queries that have the most general `ALL` as their breadth-first pattern were left out. In such cases the GrInded corpus takes longer to query than the baseline because of the overhead that results from opening and closing so many databases.[7]

In the online interface of GrETEL we remedy the possibility of an unspecified pattern by making sure the suitable version of the corpus is queried, regular or GrInded. In practice this means that we take the XPath that is based on the user's input, and try to transform it into a breadth-first pattern, as discussed above. Sometimes a pattern cannot be generated, e.g. when the XPath does not contain at least two levels, or when the top node is unspecified (cf. supra `ALL`) and no pattern can be generated from its children (because they don't have specified dependency relationships (`rel` attributes) for instance). When this is the case and the pattern would be simply `ALL`, our system recognizes that the input query is too underspecified to work well with GrInd so we query the regular version of the corpus.

This begs the question which XPath structures can be used to query the GrInded treebanks. As said before, the back-end system will try to create a breadth-first pattern from a given XPath. If it cannot do that, the regular version of the treebank is used. The only requirement for a given XPath query to be transformed into a valid pattern is, in its most basic form, that the topmost node has a specified syntactic category (`cat` attribute e.g. `np`) and at least one child with a dependency relation (`rel` attribute, e.g. `hd`). With this information, a breadth-first pattern can be generated (e.g. `nphd%`) and the search space has already been greatly reduced. Apart from that, any valid XPath 3.0 expression can be used in the query.[8] What this means is that no restrictions have to be made to XPath expressions used to mine the GrInded treebanks except that the first level has to have a `cat` attribute (without negation and disjunction) and the second level a `rel` attribute (without negation and disjunction). The rest of the XPath structure can contain any valid XPath properties such as negation, conjunction, disjunction, quantification, and XPath's built-in functions. In the future, however, we would also like to extend the script that generates the breadth-first

7. It would of course be possible to devise a form of indexing that focuses on lexical queries, such as a lookup system that allows to retrieve sentence numbers by lemma.

8. As of BaseX 8.0, XPath 3.1 is also supported. We ran our benchmarks on the older version 7.9.

patterns from a given XPath structure to handle negation and disjunction for the topmost category or second-level relations (cf. Section 6).

Finally it should be noted that the database system BaseX offers an indexing system on its own as well[9], as most database systems do. Some indexes are created automatically and cannot be dropped; they are called structural indexes (name, path, and document indexes). These indexes are always created and present. Value indexes, on the other hand, can be created or dropped by the database manager. Available value indexes are the automatically created indexes `text` and `attribute`, and a `token` index (since version 8.4) and a `fulltext` index. `text` is not relevant for our GrInded treebanks, nor for treebanks without text nodes in general, because it would generate an index for the text nodes in a document. A text node is a node in XML that has text inside of it (e.g. Example 8a). However, in our data (as output by the Alpino parser) there are no text nodes inside the dependency trees. The text of terminal nodes are given as `word` attributes of text-less element nodes (e.g. Example 8b). The XML generated by Alpino does provide a `sentence` text node in the original treebank, placed after the dependency tree itself, which contains the sentence as a flat string. If it would be required to query this node (for a string-based query) then a `text` index for this node would be useful in the regular corpus. However, we are focusing on querying the treebank and its dependency structures, and not the corpus as a text-only document, so this index is not relevant for us. Note that the `sentence` tag is not used in the GrInd process and not available in the GrInded treebank itself. Therefore, we can drop this index to save some disk space when importing into BaseX. A `fulltext` index is not useful for us in the same way as a `text` is not. The difference between the two is that full-text index keeps track of all the tokens of the XML file's text nodes which is useful to check to see if an element (or its children) contains a string value. Again, this is not useful for our data.

(8) a. `<node pt="n">man</node>`
 b. `<node pt="n" word="man"/>`

In version 8.4 of BaseX a `token` index has been introduced. Even though we have only used version 7.9 of BaseX, it could be interesting for future reference to look at this index option. In some varieties of XML, of which HTML is probably the most well-known, it is allowed to place multiple, space-separated items as attribute values. These individual values are called tokens. An obvious example is how multiple classes can be added to a single HTML element (Example 9a). A sensible example of tokens in a linguistic environment is given in Example 9b, which could represent an XML dictionary entry for the word `sheep` that can be singular as well as plural. However, for the Alpino treebanks we use, it is never the case that there are multiple tokens for an attribute. In other words, a `token` index is not useful either.

(9) a. `<div class="row flex animals"/>`
 b. `<node count="sg pl" word="sheep"/>`

The `attribute` index, however, is of paramount importance. In Alpino XML all node features are represented as XML attributes (similar to Example 8b but much more extended). This is not different in the result of GrInd. An `attribute` index on top of our own GrETEL Indexing system ensures we can still use the benefits of BaseX's indexing system on the reduced search space that results from our own GrETEL Indexing. So it is not a case of choosing between one or the other; BaseX's indexing, specifically its `attribute` index, works on top of our own indexing mechanism.

4.3 Benchmark set-up

The treebank consists of XML files, so we opted to use the native XML database BaseX, as mentioned many times before. We used BaseX version 7.9. The SoNaR treebank consists of 25 components

9. http://docs.basex.org/wiki/Indexes

containing different written text genres such as magazines, websites, Wikipedia articles, newspapers, auto cues, and so on. We aimed to test the influence of the GrInd procedure on the entire treebank, but three components (WR-P-E-L, WR-U-E-A, WR-U-E-D) are not included in the benchmark because they lack lemma tags and the Dutch CGN-D-COI POS tags (`pt`). They are written in italics in table 1. In case of the 22 other components, we loaded the baseline and the GrInded version of each component into BaseX, and then ran each of the 87 queries on the two versions. We recorded the time it took to finish a query (i.e. until all results are found and all required databases have been queried), and we also kept track of the amount of results that were actually found, just to make sure that the versions provide the same results. As already briefly mentioned in subsection 4.2, all queries were run five times for component WR-P-E-C to investigate caching.

All queries for every component (GrInded or not) were run on the same server (Intel Xeon E5-1650 v2 @ 3.50GHz (6 cores), 64 GB of main memory) to ensure the same hardware fundamentals. We used BaseX version 7.9 on the Linux-based operating system CentOS 6.

In a second, independent hardware set-up, we kept track of the creation time of the GrInded components. We also wanted to record the time it took to import the created XML files into BaseX to compare the baseline to the experiment. However, this attempt could not be completed due to time limitations caused by degrading performance when creating millions of files in a single directory. BaseX requires all databases created by a server instance to be set in one directory. Because we create millions of databases for even a single component, performance decreases and the time it takes to complete the import process takes a long time. This is not an issue specific to GrInd or BaseX but to how most file systems work, in particular NTFS on Windows even though a similar issue may arise on other systems. It may be clear that on a large scale, file creation in a single directory does not happen in linear time.[10] A solution would be to split up the data among different BaseX server instances but up to now we have not tested this. When creating the GrInded XML files, however, we do split them up into directories. One directory per possible syntactic category (`cat` attribute). Each XML file, corresponding to a breadth-first pattern, is then put into the directory that is the same as the category of its topmost head, e.g. npdet%lid$_h$d%n.xml is placed inside `np/`. From this directory structure it follows that the performance decrease as described above is less noticeable when GrInding because all files are distributed over a number of subdirectories.

The hardware of the second system that measured the time it takes to create the GrInded components consists of a Windows 10 64 bit machine, with an Intel i7-7700K processor @ 4.50GHz (4 cores), 32 GB of main memory, and a 960 GB SSD.

5. Results

To give a general idea of the impact of GrInd, we first describe a summary of the results of all components. Then we focus on three specific components: WR-P-E-C (551,119 sentences), WR-P-E-H (40,715 sentences), and lastly WR-P-P-H (5,475,556 sentences). For component WR-P-E-C we also investigate a possible caching effect, so we will discuss that component before the other two. Finally, we will discuss the time it takes to create the GrInded components in section 5.5.

5.1 General

Tables 2 and 3 present the results of the benchmark experiment, concerning speed and size respectively which are in turn visualised in Figures 5 and 6. In this section, we will first discuss the speed gain and then the size increase.

The results in Table 2 and in Figure 5 clearly show the positive effect of the GrInding process. The speed gain goes up to 1180% for the largest component WR-P-P-G that consists of 14,974,007 sentences. In absolute numbers, this means that for our benchmark the median query in the experiment

10. Cf. `https://stackoverflow.com/a/291292/1150683` for more information on degrading performance when working with millions of files in a single directory.

finishes 1841 seconds (i.e. more than half an hour) faster than the median query of the baseline. The graph indicates that the size of a component plays a large role in the speed gain that can be achieved. This is no surprise: the larger a treebank is, the more gain can be made by reducing the search space efficiently.

component	sentences	base speed (sec)	GrInd speed (sec)	speed gain (sec)	speed gain (%)
WRPEE	115	0.02	0.09	0	-78%
WRPPD	2185	0.15	0.32	0	-53%
WRPEK	8616	0.47	0.78	0	-40%
WRPPJ	16,938	1.09	0.78	0	40%
WRPEF	18,373	1.21	1.24	0	-2%
WRPPE	19,077	0.88	0.77	0	14%
WRUEE	23,498	1.26	1.13	0	12%
WRPEH	40,715	1.60	0.38	0	319%
WSUTB	44,866	2.47	1.85	1	34%
WRPPC	74,921	4.36	2.52	2	73%
WRPPK	93,565	7.83	4.45	3	76%
WRPEI	205,037	11.09	4.46	7	149%
WRPPI	387,534	32.05	14.48	18	121%
WRPEC	551,343	30.41	8.87	22	243%
WRPPF	650,509	37.89	12.68	25	199%
WRPEJ	1,355,061	80.34	36.12	44	122%
WRPPB	1,710,131	94.83	45.5	49	108%
WSUEA	2,163,661	103.49	63.77	40	62%
WRPEG	3,925,834	114.01	42.47	72	168%
WRPEA	4,396,361	555.86	156.61	399	255%
WRPPH	5,476,086	407.17	46.78	360	770%
WRPPG	14,974,007	1997.52	156.04	1841	1180%

Table 2: Overview of speed gain for all components (sorted by number of sentences)

On small treebanks components (< 19,000 sentences) the GrInded version performs worse than the baseline, as expected. Since those treebanks are already small, and the baseline query speed is in general already rather fast, the overhead of opening multiple databases cripples the overall performance. However, it is important to take a look at the absolute numbers as well. For the components where GrInding performs worse than the baseline version (WR-P-E-E, WR-P-P-D, WR-P-E-K, WR-P-E-F), the absolute difference between the baseline and the experiment is never more than 0.31 seconds, a negligible difference. In the largest negative relative difference between the two treebank variants, where the baseline is 78% faster than the experiment, the absolute difference is only 0.07 seconds (0.02 vs. 0.09 seconds). In other words, if the GrInded variant of a treebank performs worse than the baseline the difference is always less than a second, which is insignificant from a usability perspective. In contrast, if the query time adds up to a significant duration (from one second up to half an hour), the GrInded version increasingly outperforms the baseline version. These cases are the ones that matter: there is no difference in user-friendliness when a user has to wait 0.15 seconds compared to 0.32 seconds, but if a task completes in 1998 seconds (33 minutes

and 18 seconds) instead of 156 seconds (2 minutes and 36 seconds) this makes a huge difference in a real-world scenario.

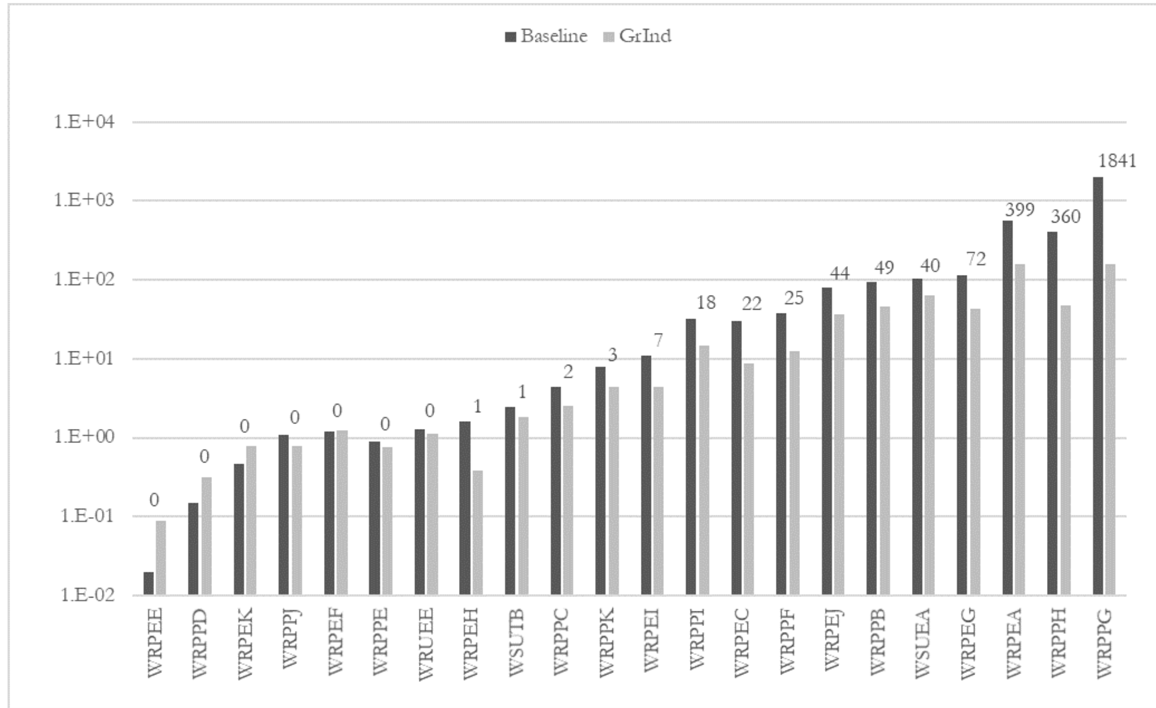


Figure 5: Graph indicating the query time of the GrInded components. Y-axis in log seconds. Values given above the data bars indicate the absolute difference in query time between the two treebank variants (in seconds). X-axis sorted by component size (number of sentences, small to large).

Since GrInd creates one XML file per subtree pattern, we expected the GrInded treebank size to be much larger than the baseline. Figure 6 and the values in Table 3 show that the size of the treebank grows immensely indeed. However, the relative size increase decreases if the size of the components increases, which is clear in the last column of said table. For instance, for the smallest component WR-P-E-E (115 sentences) the baseline is around 1 MB in size but the GrInded version is almost 19 MB, indicating a size increase of 1704%. WR-P-E-H (40,715 sentences) normally is 224 MB in size, but when GrInded it is 1357 MB; an increase of 505%. The largest component, WR-P-P-G, containing 14,974,007 sentences, ticks in at 102,603 MB (1̃02 GB) for the baseline, and 461,954 MB (4̃62 GB) for the GrInded variant, which comes down to a relative increase of 350%. We can explain this relative decrease in size as follows. Even though there is more data to GrInd, there is only a finite amount of breadth-patterns that can be generated. This means that the chance for a subtree to match a pattern from another subtree increases, and that no new files need to be created. As a consequence, the amount of files that *only* contain “includes” stagnates, as these files tend to get populated with subtrees as well. The chance that an include has to be created for a pattern that only occurs once in the corpus decreases. Put differently, due to a finite set of possibilities of patterns given the parameters in 1, the size increase stagnates as the baseline size of the treebanks increases.

component	sentences	base size (MB)	GrInd size (MB)	size increase (MB)	size increase (%)
WRPEE	115	1.04	18.76	18	1704%
WRPPD	2185	16.55	205.96	189	1144%
WRPEK	8616	67.38	664.98	598	887%
WRPPJ	16,938	153.56	1333.78	1180	769%
WRPEF	18,373	163.45	1299.24	1136	695%
WRPPE	19,077	117.82	840.34	723	613%
WRUEE	23,498	171.18	1311.24	1140	666%
WRPEH	40,715	224.28	1357.15	1133	505%
WSUTB	44,866	332.61	2637.79	2305	693%
WRPPC	74,921	593.59	4167.51	3574	602%
WRPPK	93,565	1059.30	8435.46	7376	696%
WRPEI	205,037	1514.64	8857.22	7343	485%
WRPPI	387,534	4299.64	30,460.45	26,161	608%
WRPEC	551,343	4133.80	22,517.73	18,384	445%
WRPPF	650,509	5148.50	32,772.14	27,624	537%
WRPEJ	1,355,061	11,021.06	55,431.75	44,411	403%
WRPPB	1,710,131	12,892.71	70,861.24	57,969	450%
WSUEA	2,163,661	13,729.76	62,516.04	48,786	355%
WRPEG	3,925,834	14,287.14	46,975.34	32,688	229%
WRPEA	4,396,361	27,749.84	137,737.32	109,987	396%
WRPPH	5,476,086	44,860.68	225,906.22	181,046	404%
WRPPG	14,974,007	102,603.05	461,954.15	359,351	350%

Table 3: Overview of the size increase for all components (sorted by number of sentences)

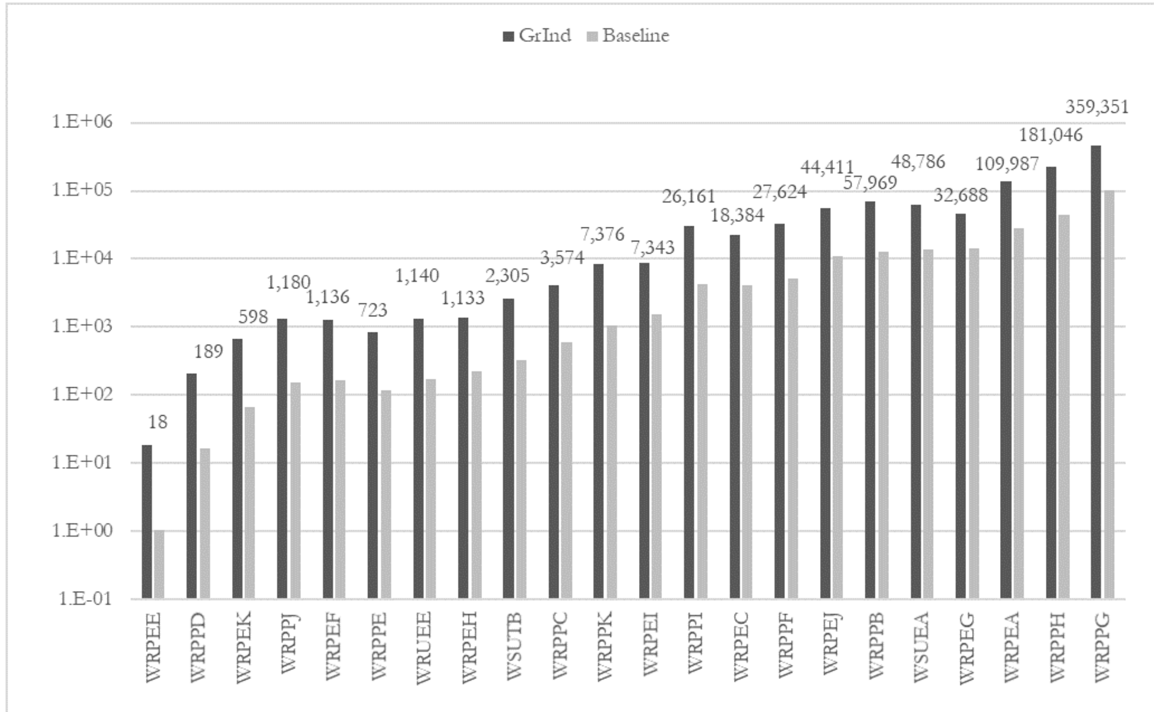


Figure 6: Graph comparing the size of the GrInded and baseline components. Y-axis in log megabytes. Values given above the data bars indicate the absolute difference in size between the two treebank variants (in MB). X-axis sorted by component size (number of sentences, small to large).

In sum, it is clear that the GrInd process indeed leads to faster query times. In turn, the disk usage increases quite a lot as well. The benchmark experiment shows that the relative increase in speed is bigger, and the relative increase in disk space is smaller if the size of a component increases. In the next sections we will briefly discuss three components in more detail, i.e. WR-P-E-C, WR-P-E-H, and WR-P-P-H.

5.2 Results for WR-P-E-C

Table 4 presents the results of the benchmark experiment for the treebank component WR-P-E-C. The results are divided into three columns: one column for all hits, and two columns representing a subset, i.e. those where no results were found, and those where more than zero results were found. Furthermore, the query time for each of the five iterations is shown for the baseline as well as the GrInd version. This is done to show the possible caching effect: if the median result of the first run across all queries is noticeably higher than the medians for the following iterations, a caching effect is present.

		All hits	>0 hits	0 hits
Baseline	*Run 1*	**30.4**	29.8	30.7
	Run 2	29.8	29.5	30.1
	Run 3	29.8	29.3	30.2
	Run 4	30.0	29.5	30.2
	Run 5	29.8	29.6	30.0
GrInd	*Run 1*	**8.9**	16.5	4.7
	Run 2	3.0	6.0	1.3
	Run 3	3.0	5.9	1.3
	Run 4	3.8	6.9	1.3
	Run 5	3.6	6.6	1.2

Table 4: Benchmark results on WR-P-E-C (in seconds). The 87 test queries were run five times to determine a caching effect.

The results in Table 4 give a clear indication of a a caching effect in the GrInded version, and a lot less prominent one in the baseline. If an XPath structure is looked up for the first time it takes much longer to complete than for the next iterations, implicating that the results are cached one way or another. This behavior is only superficially visible in the baseline, at least on a much smaller scale than in the GrInded version. The reason for this is not clear. We asked the BaseX team whether they had any idea how this was possible, but without a full test case they could not answer the question either. Maybe in the future, a collaboration with them will lead to an answer.

Another major difference between the baseline and the GrInd results is that the GrInded version performs incredibly fast when zero hits are found. Even though this may seem trivial, it is not. Instead of needing to search through the entire corpus to make sure no hits are found, BaseX can quickly go through the small defined search space. When zero hits are found this often means that quite a specific (and/or long) pattern is queried, which either simply does not exist as a database, or only has a few included databases. This is not always the case though. Some queries had to go through more than 4000 databases and returned zero hits nonetheless. Still, considering the median speed, finalizing a query which returns zero hits is much quicker in the GrInded version compared to the baseline. In a real-world application this means that instead of waiting for half a minute to

be presented with zero results, a user only needs to wait for 5 seconds to realise that no hits were found and that the input query might need modification.

Table 4 presents another interesting fact. The amount of hits that are found does not influence the search time in any way that is different for the GrInded or the baseline version, but the number of databases that are queried to find all results, be they many or few, does play a big role in how fast the GrInded corpus can actually be queried. In the baseline version, this number is fixed: there are 56 treebank parts in the baseline version of component WR-P-E-C, which all need to be searched through for every query. For the GrInded version of the corpus, the amount of databases differs for each query. The data suggests that around 500 includes a turning point takes place: the baseline version keeps its steady pace of querying in circa 30 seconds, but when the amount of includes gets higher than 500 in the GrInded version, more time is needed to find all results in the GrInded component than in the baseline. This is due to the overhead of opening and closing database connections. The sequential processes of opening a database connection and sending a new query causes an overhead that becomes noticeable as soon as the includes are high in number. This factor was expected, but now it is measured that the turning point is at around 500 for a corpus this size.

5.3 Results for WR-P-E-H

We expected the GrInding process to have more effect on larger corpora. Nevertheless it is interesting to see how it performs on smaller ones as well. WR-P-E-H contains 40,715 sentences, which is fourteen times smaller than WR-P-E-C. Its contents are teletext pages. The benchmark results are given in Table 5.

	All hits	>0 hits	0 hits
Baseline	**1.6**	1.6	1.6
GrInd	**0.4**	0.7	0.2

Table 5: Benchmark results on WR-P-E-H (in seconds)

The effect of GrInding on a smaller component should not be underestimated. With a mean speed of 0.4 seconds the GrInded version is still faster than the baseline component. Again, in cases where zero hits are found, the GrInded component is very fast. When looking at the actual search times, it becomes clear that for WR-P-E-H the turning point of advantage lies at around 120 includes. If BaseX has to go through more includes, searching is not as fast as the baseline. In practice, however, GrInding a small component might not be worth the time and processing power. 1.6 seconds to wait for results from the baseline version is not a long time, and considering user-friendliness, this is not unacceptable. The results for WR-P-E-H amplify the earlier conclusion that GrInd is mainly suitable for large treebanks.

5.4 Results for WR-P-P-H

WR-P-P-H is the second largest component of SoNaR, containing 5,475,556 sentences from periodicals and magazines. Because of its size, it is expected that the impact of the GrInding procedure is very noticeable. The benchmark results for this component are presented in Table 6.

As expected, GrInding is very effective on large corpora. While the baseline performs more or less equally in all cases, the GrInded version shows a large difference when returning 0 hits or not. Considering all queries (no matter the amount of hits) the execution time for the GrInded version is 46.8 seconds, which highly contrasts with the baseline that needs 407.2 seconds to complete. The

	All hits	>0 hits	0 hits
Baseline	**407.2**	413.2	404.6
GrInd	**46.8**	51.9	12.7

Table 6: Benchmark results on WR-P-P-H (in seconds)

experiment shows that the GrInded component should be faster up to 9000 includes. Otherwise, the baseline will finish the search process more quickly.

5.5 Creation time

As mentioned before, we executed a second benchmark at a later time to measure the time it takes to create the GrInded treebanks. Unfortunately, we were not able to GrInd the largest component WR-P-P-G this second time due to unexpected hardware failure. However, the timings for the other components are presented in Table 7 and visualised in Figure 7.

component	sentences	total (s)	total (d hh:mm:ss)
WRPEE	115	23	0:00:23
WRPPD	2185	428	0:07:08
WRPEK	8616	2187	0:36:27
WRPPJ	16,938	4600	1:16:40
WRPEF	18,373	4733	1:18:53
WRPPE	19,077	3661	1:01:01
WRUEE	23,498	5469	1:31:09
WRPEH	40,715	6001	1:40:01
WSUTB	44,866	10,312	2:51:52
WRPPC	74,921	13,642	3:47:22
WRPPK	93,565	24,442	6:47:22
WRPEI	205,037	23,076	6:24:36
WRPPI	387,534	58,584	16:16:24
WRPEC	551,343	82,924	23:02:24
WRPPF	650,509	99,446	1 day 03:37:26
WRPEJ	1,355,061	117,778	1 day 08:42:58
WRPPB	1,710,131	323,148	3 days 17:45:48
WSUEA	2,163,661	121,168	1 day 09:39:28
WRPEG	3,925,834	103,766	1 day 04:49:26
WRPEA	4,396,361	359,748	4 days 03:55:48
WRPPH	5,476,086	588,153	6 days 19:22:33

Table 7: Overview of the time needed to created the GrInded treebanks (sorted by number of sentences)

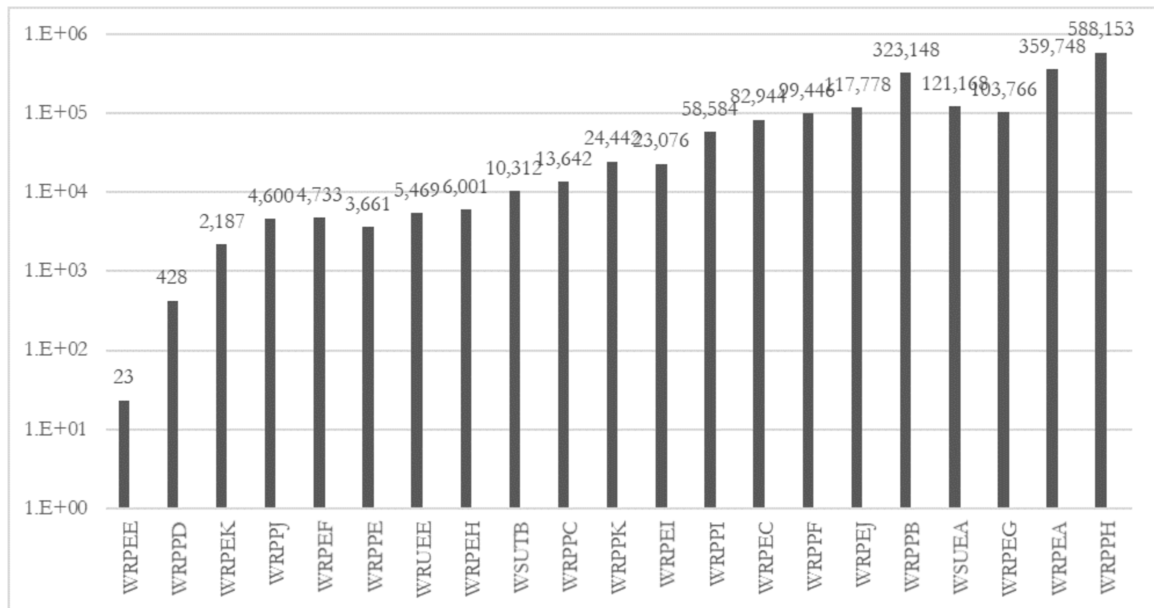


Figure 7: Graph indicating the creation time of the GrInded components. Y-axis in log seconds, absolute values (in seconds) given above the data bars. X-axis sorted by component size (number of sentences, small to large).

It may be clear the process takes a long time. This is not surprising; every sentence (tree) has to be processed recursively to find all possible subtrees derived from it. It is a combinatorial explosion. It follows that the number of sentences is not the sole factor determining the duration of the GrInd process. More important is the size of the dependency trees derived from the sentences. The broader or deeper a tree, the longer it takes to GrInd it.

6. Conclusion and future work

We presented GrETEL Indexing, or GrInding, an indexing mechanism that reduces the search space before an actual treebank search is performed. To show the improvement in query time, we discussed the results of a benchmark experiment, testing the effect of the GrInding procedure on the SoNaR-500 treebank.

We show that the process of GrInding indeed has an effect on the search speed, as the theoretical assumptions in Vandeghinste and Augustinus (2014) foresaw. The effect is enormous when no hits are found, and very noticeable in all other cases. It is important, though, that GrInding only has a positive influence up to a specific amount of includes (dependent on the size of the corpus). The benchmark results show that if the search engine has to go through more than 120 (for WR-P-E-H), 500 (for WR-P-E-C), or 9000 (WR-P-P-H) includes, the overhead caused by opening all these database connections turns the tables, making the GrInded component slower to query than the baseline. This means that GrInding works very well if one is looking for well-defined and specific XPath structures. If a GrInded treebank is queried with a clear XPath structure which leads to a particular breadth-first pattern, the effect of GrInd clearly emerges, but if one tries to look up very general constructions, the positive effect of GrInd diminishes. Luckily our back-end system is able to detect when a well-defined pattern could not be generated, in which case the regular version of the treebank is queried. Smaller treebanks (< 19,000 sentences) that can be queried quickly without GrInding would generally not benefit from the process. Relatively speaking, GrInding performs

worse in these cases, even though the absolute difference in query times is hardly noticeable by a user (less than a second in our benchmark). A negative side-effect of the process is that if a treebank is GrInded, it increases a lot in size so additional disk space is needed. The process itself takes a long time to run because it is a recursive combinatorial process running through every dependency tree, representing a sentence.

In general, we can conclude that large treebanks greatly benefit from GrInding. The larger the treebank, the bigger the relative improvement in query time. When working with very large treebanks, GrInding can save dozens of minutes per input query, allowing researchers to be more productive in their corpus analysis.

At the moment of writing we have nearly finished work on GrInd as a Perl module. The module allows users to parse their own XML into a structure built out of breadth-first patterns they define themselves; the user can choose how the pattern is built, i.e. which XML attributes of the first and second level of each subtree are used to create the pattern. Many other options such as separators for the pattern, using default values in a pattern, and encoding are available. The parameters as described in Item 1 in Section 3 can be chosen by the user as well. The module is open-source.[11] Testing the module and giving feedback is possible on the project's GitHub page[12] and highly encouraged.

Additionally, as expressed in 4.2, we also would like to improve the breadth-first generation system to include XPath that has negation or disjunction in its topmost category or second-level dependency relation. To be able to exclude patterns from the generated pattern and subsequently possibly generating multiple patterns at once, it is required to know which possible `cat` and `rel` values are available. For instance, if a user wants to look for an XPath structure whose topmost node is not a noun phrase, we have to know which other possible `cat` values apart from `np` are available in the treebank to make sure we generate breadth-first patterns with existing syntactic categories. Alternatively, it might be possible to generate the breadth-first pattern as if the negation was not present, and then do some string comparison with the available databases (patterns) to exclude the given category on a substring level. Further research is necessary to create the most efficient solution to this problem.

In future work we will use aforementioned module to GrInd the parallel treebanks to use in Poly-GrETEL (Augustinus et al. 2016).[13] Currently this tool includes Dutch and English syntax trees from Europarl (Koehn 2005), and it will be extended with German soon.

7. Acknowledgements

This research is done in the context of an internship of the Masters of Artificial Intelligence program at KU Leuven and the SCATE project, funded by the Flemish Agency for Innovation through Science and Technology (IWT SBO, Project Nr. 130041)

References

Augustinus, Liesbeth, Vincent Vandeghinste, and Frank Van Eynde (2012), Example-Based Treebank Querying, *Proceedings of the 8th International Conference on Language Resources and Evaluation (LREC 2012)*, Istanbul, pp. 3161–3167.

Augustinus, Liesbeth, Vincent Vandeghinste, and Tom Vanallemeersch (2016), Poly-GrETEL: Cross-Lingual Example-based Querying of Syntactic Constructions, *Proceedings of the 10th Interna-*

11. `https://github.com/CCL-KULeuven/grinding`. Note that the module is not completely finished yet. Documentation still has to be written, and an encoding option is not yet present. For now, only UTF-8 encoded XML files are supported.
12. `https://github.com/CCL-KULeuven/grinding/issues`
13. `http://gretel.ccl.kuleuven.be/poly-gretel`

tional Conference on Language Resources and Evaluation (LREC 2016), Portorož, pp. 3549–3554.

Augustinus, Liesbeth, Vincent Vandeghinste, Ineke Schuurman, and Frank Van Eynde (in press), GrETEL: A tool for example-based treebank mining, *in* Odijk, Jan and Arjan van Hessen, editors, *CLARIN in the Low Countries*, Ubiquity Press, London.

Koehn, Philipp (2005), Europarl: A Parallel Corpus for Statistical Machine Translation, *Proceedings of MT Summit X*, Phuket, pp. 79–86.

Martens, Scott (2013), TüNDRA: A Web Application for Treebank Search and Visualisation, *Proceedings of the 12th International Workshop on Treebanks and Linguistic Theories (TLT12)*, Sofia, pp. 133–144.

Meurer, Paul (2012), INESS-Search: A search system for LFG (and other) treebanks, *Proceedings of the LFG'12 Conference. LFG Online Proceedings*, Stanford, pp. 404–421.

Odijk, Jan (2015), Linguistic Research with PaQu, *Computational Linguistics in the Netherlands Journal* **5**, pp. 3–14.

Oostdijk, Nelleke, Martin Reynaert, Véronique Hoste, and Ineke Schuurman (2013), The Construction of a 500-Million-Word Reference Corpus of Contemporary Written Dutch, *in* Spyns, Peter and Jan Odijk, editors, *Essential Speech and Language Technology for Dutch. Results by the STEVIN programme*, Springer, pp. 219–247.

Reynaert, Martin, Matje van de Camp, and Menno van Zaanen (2014), Opensonar: user-driven development of the sonar corpus interfaces, *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: System Demonstrations*, Dublin City University and Association for Computational Linguistics, pp. 124–128. http://aclanthology.coli.uni-saarland.de/pdf/C/C14/C14-2027.pdf.

van der Wouden, Ton, Heleen Hoekstra, Michael Moortgat, Bram Renmans, and Ineke Schuurman (2002), Syntactic Analysis in the Spoken Dutch Corpus (CGN), *Proceedings of the 3rd International Conference on Language Resources and Evaluation (LREC 2002)*, Las Palmas, pp. 768–773.

van Noord, Gertjan (2006), At Last Parsing Is Now Operational, *Proceedings of TALN*, pp. 20–42.

van Noord, Gertjan, Gosse Bouma, Frank Van Eynde, Daniël de Kok, Jelmer van der Linde, Ineke Schuurman, Erik Tjong Kim Sang, and Vincent Vandeghinste (2013), Large Scale Syntactic Annotation of Written Dutch: Lassy, *in* Spyns, Peter and Jan Odijk, editors, *Essential Speech and Language Technology for Dutch. Results by the STEVIN programme*, Springer, pp. 147–164.

Vandeghinste, Vincent and Liesbeth Augustinus (2014), Making Large Treebanks Searchable. The SONAR case, *Proceedings of the LREC 2014 2nd workshop on Challenges in the management of large corpora (CMLC-2)*, Reykjavik, pp. 15–20.

Vandeghinste, Vincent and Scott Martens (2010), Bottom-up transfer in Example-based Machine Translation, *in* Ivon, François and Viggo Hansen, editors, *Proceedings of the 14th International Conference of the European Association for Machine Translation (EAMT-2010)*, Saint-Raphael.