Build your own treebank

Daniël de Kok, Dörte de Kok, Marie Hinrichs
University of Tübingen
Wilhelmstr. 19, 72074 Tübingen
daniel.de-kok@uni-tuebingen.de, doerte.de-kok@uni-tuebingen.de, marie.hinrichs@uni-tuebingen.de

1. Introduction

Large automatically annotated treebanks can be a useful resource to estimate the distribution of lexical or syntactic phenomena in a language. For example, Bouma and Spenader (2009) use an automatically annotated version of the Twente News Corpus to study the distribution of weak and strong object reflexives in Dutch, Hinrichs and Beck (2013) use among other corpora the automatically annotated TüPP-D/Z corpus to study auxiliary fronting in German, and Samardžić and Merlo (2012) use an automatically parsed version of the Europarl corpus to study causitive alternation.

In the computational linguistics community, applications such as GATE (Bontcheva et al., 2004) and WebLicht (Hinrichs et al., 2010) have been developed to make natural language processing tools available to the wider linguistic community. These applications offer a user-friendly interface, wherein the user can pick a chain of annotation tools and execute it on a text. The results of syntactic analysis can be searched and visualized using treebank search tools such as TIGERSearch (Lezius, 2002), ANNIS (Zeldes et al., 2009), INESS-Search (Meurer, 2012), Dact (Van Noord et al., 2013) and Tündra (Martens, 2013). Ideally, annotation and search are integrated, such as in the case of INESS (Rosén et al., 2012) or WebLicht plus Tündra.

However, current tools are often suboptimal for the construction and exploitation of large treebanks. In order to parse large amounts of text, it needs to be chunked into smaller parts and distributed among a large number of CPUs. This usually requires technical know-how of the parser being used and cluster batch management tools. In addition, the treebank search tools were often developed for smaller, manually annotated corpora, and do not scale to larger corpora.

In this paper, we present our latest work on scalability in WebLicht and Tündra, with the explicit goal of making construction and use of large automatically annotated treebanks available to the linguistics community. We then address the parts that still require development and relate our work to existing work, particularly INESS and GATE.

2. WebLicht

2.1. Introduction

WebLicht (Hinrichs et al., 2010) is an execution environment for natural language processing pipelines. It uses a Service Oriented Architecture (SOA) (Natis, 2003), which means that distributed, independent, RESTful web services are combined to form a chain for text analysis. Chains are constructed and executed using the web service chainer. This component constructs chains based on the profile of the input data and descriptions of web services in the form of CMDI metadata. The role of the chainer is three-fold: (1) it suggests which services can be added to the chain; (2) it checks whether a chain is valid; (3) it orchestrates the execution of the chain, which is done by sequentially sending POST requests to the services, where the body of the request to service $n + 1$ is the response of service n. WebLicht webservices use the Text Corpus Format (TCF) (Heid et al., 2010) as their interchange format, although services that perform conversions to and from TCF are also provided.

One important advantage of WebLicht's SOA architecture is that it is very easy to add a new service: a CLARIN center hosts the service and adds the service metadata to its repository.

2.2. Scalability

Two aspects of scalability in the WebLicht infrastructure can be considered: the number of concurrent users and the size of the inputs that it can handle. The number of users that can be served simultaneously can be increased by introducing more concurrency in a service. The processing speed of large inputs can be improved by introducing parallel processing. Obviously, there is interaction between both types of scalability in that both compete for CPU time.

Since most WebLicht services are implemented using JAX-RS and deployed in a Java servlet container, they provide the first type of scalability by allowing concurrent requests. However, services typically do not do any resource management. Consequently, sending a large number simultaneous requests can lead to resource starvation. Additionally, these services do not perform parallel processing at the request level.

Due to WebLicht's service-oriented architecture, it is difficult to address scalability in the web service chainer. In order to do so, it would need to know how to split up an input into chunks, what the properties of the wrapped text analysis software are, and what resources are available to a web service. This would complicate the architecture of WebLicht significantly, for a small number of services which have heavy processing requirements. Instead, we opted to provide building blocks for scalable services, which can be used to make existing services more scalable when necessary, without making any changes to the WebLicht architecture. In the next section, we discuss these building blocks, which form a distributed task queue.

2.3. Distributed task queue

The principle of a task queue is simple: a client can post tasks on the queue, while workers pop tasks off the queue and perform them. In a distributed task queue, different physical machines can act as workers. The distributed task queue that we use, Jesque[1] is a small wrapper around Redis.[2] Redis is a key-value store that can store strings, lists, sets, ordered sets, hash tables, and HyperLogLogs as values. Since Redis provides commands to push and pop items from both sides of a list, Redis key-value pairs can act as a distributed task queue, where the key is the queue name and the list-typed value the queue. A task is queued by pushing a value to the left side of the list and a task can be taken from the queue by a worker by popping the rightmost list element.

Using such a queue, we can easily obtain both request concurrency and parallelization within a request. If two requests are made at the same time, both lead to creation of tasks that are put in the queue. If there is more than one worker, these tasks are processed concurrently. Parallelization within a request is achieved by applying a chunker to the input and creating a task for each chunk, which results in parallel processing of the chunks if more than one worker is available.

2.4. Worker crashes

The task queue described above is not resilient against worker crashes. When a worker starts processing a task, the task is no longer stored in Redis which can result in a task being lost if a worker crashes. To solve this problem, we implemented *durable queues*. We use Redis transactions to implement a new command that stores a task on an in-flight list when it is popped from the queue and handed to a worker. The worker removes its task from the in-flight list when it has successfully completed it.

If a worker is able to shutdown gracefully when it crashes, it can requeue its task and remove it from the in-flight list. Otherwise, the task will remain on the in-flight list. We run a separate process that regularly checks the in-flight lists and requeues tasks if a worker has crashed.

2.5. Fair scheduling

Another remaining problem with the task queue is that tasks for long inputs are in the same queue as tasks for short inputs. Consequently, tasks from long inputs can block those from short inputs. Jesque allows a worker to poll multiple queues. We exploit this functionality to create n different queues for a particular service. When the service gets a request to process an input, we split it in chunks of a fixed size, and create a task for each chunk in the nth queue, where $n = \log_{10}(|S|)$ and S is the set of sentences in the input. In other words, the tasks are queued based on the order of magnitude of the input. Since Jesque workers poll each queue in turn and we use a constant chunk size, we are effectively applying fair scheduling (Kay and Lauder, 1988) at the queue level.

[1]http://gresrun.github.io/jesque/
[2]http://redis.io/

2.6. Input chunking

To accommodate parallel processing, a web service should not submit a processing request as one task. First of all, because larger tasks could saturate all the workers. Second, because it does not allow parallel processing of a request if workers are idle. For this reason, we provide a library for splitting TCF in chunks at a sentence-level granularity. The library also performs the orchestration. The implementer of a service only requests submission of a TCF corpus and gets an iterator over the processed results.

2.7. Evaluation

To evaluate the performance of the distributed task queue, we benchmarked the updated Malt web service in a common deployment scenario. In this test, we install Malt workers on two VMs on the same physical machine. Both instances used Kernel Virtual Machine (KVM) virtualization and provided 4 cores (Intel Xeon X5650 2.67GHz). We then parsed Schatz im Silbersee (15,324 sentences/238,592 tokens) using 1, 2, 4, 6, and 8 cores with the updated service. When an even number of cores is used, an equal number of cores are used on each virtual machine. Figure 1 shows the results of this small experiment. We can clearly see that parsing performance scales nearly linearly with the number of cores.

Since we were happy with these results, we have deployed this architecture in our production versions of the Malt and Stanford parsers. In the meanwhile, we have used the distributed Malt parser to process 30 million sentences from the German Wikipedia.

In the future, we plan to test scenarios where the workers are on different physical machines within the same rack and a larger number of CPUs, by deploying the services in a computing cluster.

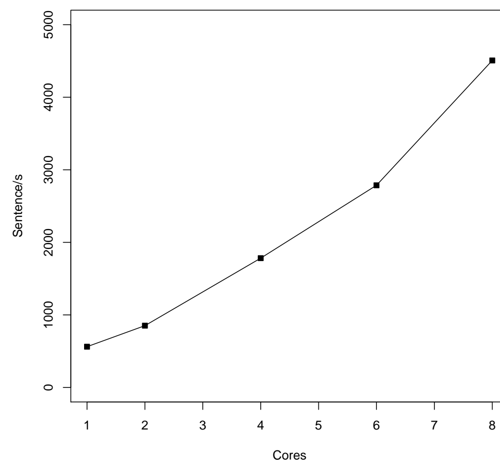

Figure 1: The number of sentences processed per second per number of cores. Measurements were made in two quad-core KVM virtual machines, running on an Intel Xeon X5650 2.67GHz.

3. Tündra

3.1. Introduction

Tündra is a web application for searching and visualizing treebanks (Martens, 2013). It supports constituency and dependency treebanks and uses the TIGERSearch query language. Tündra uses BaseX (Grün et al., 2007) to store treebanks, which is an XML database engine that supports the XQuery language. It indexes the attribute values of XML documents and performs query optimization such that indexes are accessed before processing the remainder of a query. Tündra translates TIGERSearch queries to XQuery, which is then processed by BaseX.

3.2. Scalability

Tündra was originally developed for large manually annotated treebanks, such as TüBa-D/Z (Telljohann et al., 2004).[3] We encountered some scalability issues when using larger corpora in Tündra, such as the Wikipedia treebank mentioned in Section 2.7.

3.3. Initial query processing time

The first issue encountered was that the initial processing time was often very long when running a query. BaseX can return matching nodes as they are found, allowing Tündra to show matches while the query is still running. However, if the indexes are not in the operating system's page cache, they have to be read from disk into memory. To demonstrate this overhead, we run the query `"sehen" >OBJA #o` on the databases in Table 1.[4] The results on a Mid 2011 iMac (2,7 GHz Intel Core i5, 8GB RAM) are shown in Table 2. As we can see, BaseX scales linearly on this query when the pages are in the page cache (*Hot*) and when they are not (*Cold*). However, the time to read the database into memory is very large compared to the actual running time, giving the initial delay when running the query on a large database.

Chunks	Sentences	Size (MB)
1	7214	43
10	70770	407
100	748209	4237

Table 1: Sizes of our benchmark test sets. Each chunk consists of approximately 7,000 sentences from the German Wikipedia that were parsed using the following WebLicht chain: *To TCF Converter*, *IMS Tokenizer*, *OpenNLP POS Tagger*, *IMS Morphology (RFTagger)*, *Malt Parser*, *Berkeley Parser*, and *SepVerb Lemmatizer*. The indicated sizes are the on-disk sizes of the BaseX databases.

To solve this problem, we removed instances where BaseX was directly accessed in the Tündra code base. Instead, we use a very generic treebank interface. We then follow the same approach as used by Dact (Van Noord et al., 2013) in that we provide an implementation of the treebank interface that is a concatenation of treebanks. This implementation provides a query iterator that wraps the iterators of the underlying treebanks. As a result, we can create many small treebanks that are presented as one treebank to the user. When users run a query, they will see results quickly, as the query preparation time is that of a small treebank, even for large treebanks such as Wikipedia.

Chunks	Cold (ms)	Hot (ms)	Δ
1	1506	62	1444
10	9619	502	9117
100	80424	4985	75439

Table 2: Timings for running the query `"sehen" >OBJA #o`. The cold timings indicate the processing times when the database blocks are not in the operating system's buffer cache.

3.4. Statistics view

Tündra provides a statistics view, which shows frequencies of attributes on matched nodes. For example, if the query in the previous section is executed, viewing the *lemma* attribute of the nodes bound to variable *o* will give the frequencies of lemmas that are the direct object of *sehen*. The statistics view had two problems when used on large treebanks. The first was that it only showed statistics after the query execution was finished. This is undesirable for exploratory research, since the execution of a query could take minutes or even hours. We changed the statistics view to show intermediate results while the query is running, which allows the user to quickly get an idea of the distribution of the selected attribute.

The second problem in the statistics view was that the Tündra process could run out of memory when a query was executed that resulted in a large number of hits where one of the attributes has many unique values. For instance, if we are interested in what part-of-speech follows an article, we could run a query such as `[cpos="ART"] . #w`. Since a statistics query will store the attribute-value frequencies of nodes bound to *w* and these nodes will have a wide variety of tokens, the query takes a large amount of memory.

To alleviate this problem, we apply reservoir sampling (Vitter, 1985) to queries which lead to a large number of results.[5] Reservoir sampling is an algorithm that creates a sample of size s from an unknown number of observations n, such that each observation has the probability $p = n/s$ of being in the sample. Reservoir sampling runs in $\mathcal{O}(n)$ time and $\mathcal{O}(1)$ space. Since reservoir sampling runs in constant space, we can ensure that such queries never consume all of the process heap, while giving the user reliable relative frequency estimates of attribute-value distributions.

4. Related work

This section discusses some overlapping capabilities of WebLicht and Tündra with work done in other projects.

[3] Release 9 contains 85,000 sentences.

[4] This query finds all nodes with have a direct object relation with nodes that have *sehen* as their token, lemma, category, subcategory, or part-of-speech.

[5] In the current version 100,000 matched nodes.

We limit the discussion to scalability, since scalability is the focus of this paper. We concentrate on two well-known projects: GATE and INESS.

GATE (Bontcheva et al., 2004) allows users to build and run NLP processing pipelines. The GATE approach differs from WebLicht in that it allows much more customization and retraining of individual tools. However, learning how to use GATE may be prohibitive for novice users. Since GATE normally runs on a user's own machine, they are responsible for scaling GATE themselves. Alternatively, GATE can be used as a commercial Cloud solution, but this requires the user to pay per hour of processing time.

INESS (Rosén et al., 2012) has rich support for treebank search and visualization. One particularly nice feature of INESS is its support of parallel corpora, which is not available in Tündra. INESS-Search also uses indexing and query optimization (Meurer, 2012) to process queries quickly. However, the authors do not report on results with treebanks at the scale of e.g. Wikipedia.

Whereas WebLicht allows end users to construct custom processing chains, INESS currently offers only authorized annotators a web interface and tool chain for preprocessing, parsing, and disambiguating large texts towards the construction of LFG treebanks.

5. Availability and future work

WebLicht users can immediately benefit from most of the improvements described in this paper. The Malt and Stanford services parsers hosted in Tübingen have been modified to use the distributed task queue. Since parsers are usually the heaviest services, this speeds up the processing of frequently-used parsing chains significantly. The first release of Tündra that hosts a dependency-parsed version of the German Wikipedia is also available.

One remaining problem is that WebLicht's workspaces are currently bound to a user session. While this works fine for e.g. parsing a novel or a month of newspaper text, a user cannot be expected to keep their browser tab open for days or even weeks to process very large corpora. We plan to modify WebLicht to support such scenarios as well, by allowing the user to log in at a later time to monitor progress or view the results. In the meanwhile, corpora that are too large for interactive processing can already be submitted by more technical users using WebLicht as a Service (WaaS).[6]

6. Conclusion

In this paper we have discussed recent advances in the scalability of WebLicht and Tündra. These improvements make it possible for linguists to construct and exploit large automatically annotated treebanks.

7. References

Bontcheva, K., Tablan, V., Maynard, D., and Cunningham, H. (2004). Evolving GATE to meet new challenges in language engineering. *Natural Language Engineering*, 10(3-4):349–373.

Bouma, G. and Spenader, J. (2009). The distribution of weak and strong object reflexives in Dutch. In van Eynde, F., Frank, A., Smedt, K. D., and van Noord, G., editors, *Proceedings of the Seventh International Workshop on Treebanks and Linguistic Theories (TLT 7)*, pages 103–114.

Grün, C., Holupirek, A., and Scholl, M. H. (2007). Visually exploring and querying XML with BaseX. In *BTW*, volume 103, pages 629–632.

Heid, U., Schmid, H., Eckart, K., and Hinrichs, E. W. (2010). A corpus representation format for linguistic web services: The d-spin text corpus format and its relationship with iso standards. In *Proceedings of LREC 2010, Malta*.

Hinrichs, E. and Beck, K. (2013). Auxiliary fronting in German: A walk in the woods. In *The Twelfth Workshop on Treebanks and Linguistic Theories (TLT12)*, page 61.

Hinrichs, E., Hinrichs, M., and Zastrow, T. (2010). Weblicht: Web-based LRT services for German. In *Proceedings of the ACL 2010 System Demonstrations*, pages 25–29. Association for Computational Linguistics.

Kay, J. and Lauder, P. (1988). A fair share scheduler. *Communications of the ACM*, 31(1):44–55.

Lezius, W. (2002). TIGERSearch ein suchwerkzeug für baumbanken. *Tagungsband zur Konvens*.

Martens, S. (2013). TüNDRA: A web application for treebank search and visualization. In *The Twelfth Workshop on Treebanks and Linguistic Theories (TLT12)*, page 133.

Meurer, P. (2012). INESS-Search: A search system for lfg (and other) treebanks. In Butt, M. and King, T. H., editors, *Proceedings of the LFG 2012 Conference*, pages 404–421. Stanford, CA: CSLI Publications.

Natis, Y. V. (2003). Service-oriented architecture scenario. Technical Report AV-19-6751, Gartner Inc., April.

Rosén, V., De Smedt, K., Meurer, P., and Dyvik, H. (2012). An open infrastructure for advanced treebanking. In *META-RESEARCH Workshop on Advanced Treebanking at LREC2012, Istanbul*, pages 22–29.

Samardžić, T. and Merlo, P. (2012). The meaning of lexical causatives in cross-linguistic variation. *Linguistic Issues in Language Technology*, 7:1–14.

Telljohann, H., Hinrichs, E., and Kübler, S. (2004). The TüBa-D/Z treebank: Annotating german with a context-free backbone. In *In Proceedings of the Fourth International Conference on Language Resources and Evaluation (LREC 2004)*.

Van Noord, G., Bouma, G., Van Eynde, F., De Kok, D., Van der Linde, J., Schuurman, I., Sang, E. T. K., and Vandeghinste, V. (2013). Large scale syntactic annotation of written Dutch: Lassy. In *Essential Speech and Language Technology for Dutch*, pages 147–164. Springer.

Vitter, J. S. (1985). Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57.

Zeldes, A., Lüdeling, A., Ritz, J., and Chiarcos, C. (2009). ANNIS: A search tool for multi-layer annotated corpora. In *Proceedings of Corpus Linguistics 2009*.

[6] `https://weblicht.sfs.uni-tuebingen.de/WaaS/`