Fast Query for Large Treebanks

Sumukh Ghodke[*]
[*]Department of Computer Science
and Software Engineering,
University of Melbourne
Victoria 3010, Australia

Steven Bird[*†]
[†]Linguistic Data Consortium,
University of Pennsylvania
3600 Market Street, Suite 810
Philadelphia PA 19104, USA

Abstract

A variety of query systems have been developed for interrogating parsed corpora, or treebanks. With the arrival of efficient, wide-coverage parsers, it is feasible to create very large databases of trees. However, existing approaches that use in-memory search, or relational or XML database technologies, do not scale up. We describe a method for storage, indexing, and query of treebanks that uses an information retrieval engine. Several experiments with a large treebank demonstrate excellent scaling characteristics for a wide range of query types. This work facilitates the curation of much larger treebanks, and enables them to be used effectively in a variety of scientific and engineering tasks.

1 Introduction

The problem of representing and querying linguistic annotations has been an active area of research for several years. Much of the work has grown from efforts to curate large databases of annotated text such as *treebanks*, for use in developing and testing language technologies (Marcus et al., 1993; Abeillé, 2003; Hockenmaier and Steedman, 2007). At least a dozen linguistic tree query languages have been developed for interrogating treebanks (see §2).

While high quality syntactic parsers are able to efficiently annotate large quantities of English text (Clark and Curran, 2007), existing approaches to query do not work on the same scale. Many existing systems load the entire corpus into memory and check a user-supplied query against every tree. Others avoid the memory limitation, and use relational or XML database systems. Although these have built-in support for indexes, they do not scale up either (Ghodke and Bird, 2008; Zhang et al., 2001)).

The ability to interrogate large collections of parsed text has important practical applications. First, it opens the way to a new kind of information retrieval (IR) that is sensitive to syntactic information, permitting users to do more focussed search. At the simplest level, an ambiguous query term like *wind* or *park* could be disambiguated with the help of a POS tag (e.g. `wind/N`, `park/V`). (Existing IR engines already support query with part-of-speech tags (Chowdhury and McCabe, 1998)). More complex queries could stipulate the syntactic category of *apple* is in subject position.

A second benefit of large scale tree query is for natural language processing. For example, we might compute the likelihood that a given noun appears as the agent or patient of a verb, as a measure of animacy. We can use features derived from syntactic trees in order to support semantic role labeling, language modeling, and information extraction (Chen and Rambow, 2003; Collins et al., 2005; Hakenberg et al., 2009). A further benefit for natural language processing, though not yet realized, is for a treebank and query engine to provide the underlying storage and retrieval for a variety of linguistic applications. Just as a relational database is present in most business applications, providing reliable and efficient access to relational data, such a system would act as a repository of annotated texts, and expose an expressive API to client applications.

A third benefit of large scale tree query is to support syntactic investigations, e.g. for developing syntactic theories or preparing materials for language learners. Published treebanks will usually not attest particular words in the context of some infrequent construction, to the detriment of syntactic studies that make predictions about such combinations, and language learners wanting to see instances of some construction involving words from some specialized topic. A much larger treebank alleviates these problems. To improve recall performance, multiple parses for a given sentence could be stored (possibly derived from different parsers).

A fourth benefit for large scale tree query is to support the curation of treebanks, a major enterprise in its own right (Abeillé, 2003). Manual selection and correction of automatically generated parse trees is a substantial part of the task of preparing a treebank. At the point of making such decisions, it is often helpful for an annotator to view existing annotations of a given construction which have already been manually validated (Hiroshi et al., 2005). Occasionally, an earlier annotation decision may need to be reconsidered in the light of new examples, leading to further queries and to corrections that are spread across the whole corpus (Wallis, 2003; Xue et al., 2005).

This paper explores a new methods for scaling up tree query using an IR engine. In §2 we describe existing tree query systems, elaborating on the design decisions, and on key aspects of their implementation and performance. In §3 we describe a method for indexing trees using an IR engine, and discuss the details of our open source implementation. In §4 we report results from a variety of experiments involving two data collections. The first collection contains of 5.5 million parsed trees, two orders of magnitude larger than those used by existing tree query systems, while the second collection contains 26.5 million trees.

2 Treebank Query

A tree query system needs to be able to identify trees having particular properties. On the face of it, this should be possible to achieve by writing simple programs over treebank files on disk. The programs would match tree structures using regular expression patterns, possibly augmented with syntax for matching tree structure. However, tree query is a more complex and interesting task, due to several factors which we list below.

Structure of the data: There are many varieties of treebank. Some extend the nested bracketing syntax to store morphological information. Others store complex attribute-value matrices in tree nodes or have tree-valued attributes (Oepen et al., 2002), or store dependency structures (Čmejrek et al., 2004), or categorial grammar derivations (Hockenmaier and Steedman, 2007). Others store multiple overlapping trees (Cassidy and Harrington, 2001; Heid et al., 2004; Volk et al., 2007).

Form of results: Do we want entire trees, or matching subtrees, or just a count of the number of results? Do we need some indication of why the query matched a particular tree, perhaps by showing how query terms relate to a hit, cf. document snippets and highlighted words in web search results? Do we want to see multiple hits when a query matches a particular tree in more than one place? Do we want to see tree diagrams, or some machine-readable tree representation that can be used in external analysis? Can a query serve to update the treebank, cf. SQL update queries?

Number of results: Do we want all results, or the first n results in document order, or the “best” n results, where our notion of best might be based on representativeness or distinctiveness.

Description language: Do we prefer to describe trees by giving examples of tree fragments, replacing some nodes replaced with wildcards (Hiroshi et al., 2005; Ichikawa et al., 2006; Mírovský, 2006)? Or do we prefer a path language (Rohde, 2005; Lai and Bird, 2010)? Or perhaps we prefer a language involving variables, quantifiers, boolean operators, and negation (König and Lezius, 2001; Kepser, 2003; Pajas and Štěpánek, 2009)? What built-in tree relations are required, beyond the typical parent/child, ancestor/descendent, sibling and temporal relations? (E.g. last child, leftmost descendent, parent’s following sibling, pronoun’s antecedent.) Do we need to describe tree nodes using regular expressions, or attributes and values? Do we need a type system, a pattern language, or boolean logic for talking about attribute values? The expressive requirements of the query language have been discussed at length elsewhere (Lai and Bird, 2004; Mírovský, 2008), and we will not consider them further here.

Performance: What performance is acceptable, especially as the data size grows? Do we want to optimize multiple reformulations of a query, for users who iteratively refine a query based on query results? Do we want to optimize certain query types? Are queries performed interactively or in batch mode? Is the treebank stable, or being actively revised, in which case indexes need to be easily updatable? Do we expect logically identical queries to have the same performance, so that users do not have to rewrite their queries for efficiency? Key performance measures are index size and search times.

Architecture: Is the query system standalone, or does it exist in a client-server architecture? Is there a separate user-interface layer that interacts with a data server using a well-defined API, or is it a monolithic system? Should it translate queries into another language, such as SQL (Bird et al., 2006; Nakov et al., 2005), or XQuery (Cassidy, 2002; Mayo et al., 2006), or to automata (Maryns and Kepser, 2009), in order to benefit from the performance optimizations they provide

Indexing. The indexing methods used in individual systems are usually not reported. Many systems display nearly constant time for querying a database, regardless of the selectivity of a query, a strong indicator that no indexes are being used. For example, Emu performs all queries in memory with no indexes, and several others are likely to be the same (Cassidy and Harrington, 2001; König and Lezius, 2001; Heid et al., 2004). TGrep2 (Rohde, 2005) uses a custom corpus file and processes it sentence by sentence at query execution time. Other tree query systems use hashed indexes or other types of in-memory indexes. However, a common drawback of these systems is that they are designed for treebanks that are at most a few million words in size, and do not scale well to much larger treebanks.

There are many positions to be taken on the above questions. Our goal is not to argue for a particular data format or query style, but rather to demonstrate a powerful technique for indexing and querying treebanks which should be applicable to most of the above scenarios.

3 Indexing Trees

In this section we discuss two methods of storing and indexing trees. The first uses a relational database and linguistic queries are translated into SQL, while the second uses an inverted index approach based on an open source IR engine, Lucene.[1] Relational databases are a mature technology and are known to be efficient at performing joins and accessing data using indexes. Information retrieval engines using term vectors, on the other hand, efficiently retrieve documents relevant to a query. IR engines are known to scale well, but they do not support complex queries. A common feature of both the IR and database approaches is the adoption of so-called "tree labeling" schemes.

3.1 Tree labeling schemes

Tree queries specify node labels ("value constraints") and structural relationships between nodes of interest ("structural constraints"). A simple value constraint could look for a *wh noun phrase* by specifying the `WHNP`; such queries are efficiently implemented using indexes. Structural relationships cannot be indexed like node labels. A term in a sentence will have multiple relationships with other terms in the same sentence. Indexing all pairs of terms that exist in a given structural relationship results in an explosion in the index size. Instead, the standard approach is to store position information with each occurrence of a term, using a table or a term vector, and then use the position information to find structural matches. Many systems use this approach, from early object databases such as Lore (McHugh et al., 1997), to relational representation of tree data (Bird et al., 2006) and XISS/R (Harding et al., 2003), and native XML databases such as eXist (Meier, 2003). Here, the position is encoded via node labeling schemes, and is designed so it can support efficient testing of a variety of structural relations.

A labeling scheme based on pre-order and post-order labeling of nodes is the foundation for several extended schemes. It can be used for efficiently detecting that two nodes are in a hierarchical (or inclusion) relationship. Other labeling schemes are based on the Dewey scheme, in which each node contains

[1] http://lucene.apache.org/

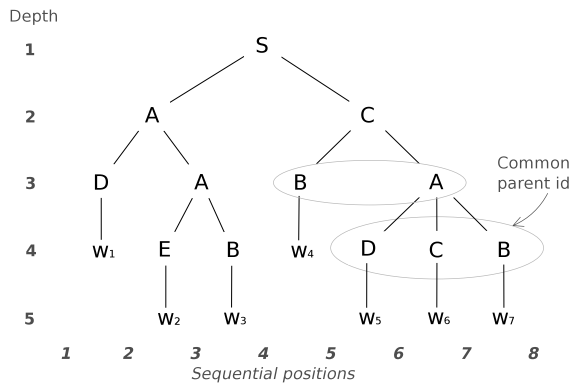

Figure 1: Generating node labels

a hierarchical label in which numbers are separated by periods (Tatarinov et al., 2002). A child node gets its label by appending its position relative to its siblings to its parent's label. This scheme can be used for efficiently detecting that two nodes are in a hierarchical or sequential (temporal) relationship.

The LPath numbering scheme assigns four integer labels to each node (Bird et al., 2006). The generation of these labels is explained with the help of an example. Figure 1 is the graphical representation of a parse tree for a sentence with 7 words, $w_1 \cdots w_7$. Let *A*, *B*, *C*, *D*, *E*, and *S* represent the annotation tags. Some nodes at different positions in the tree share a common name.

The first step in labeling is to identify the sequential positions between words, as shown beneath the parse tree in Figure 1. The left id of a terminal node is the sequence position immediately to the left of a node, while its right id is the one to its immediate right. The left id of a non-terminal node is the left id of its leftmost descendant, and the right id is the right id of its rightmost descendant. In most cases the ancestor-descendant and preceding-following relationships between two elements can be evaluated using the left and right ids alone. The sequential ids do not differentiate between two nodes where one is the lone child of the other. The depth id is therefore required in such cases and to identify the child node (depth values are shown on the left side of Figure 1). In order to check if two given nodes are siblings, the above three ids will not suffice. We therefore assign a common parent id label to siblings. These four identifiers together enable us to identify relationships between elements without traversing trees.

Node	Left	Right	Depth	Parent
A	2	4	3	2
A	1	4	2	6
A	5	8	3	8
B	3	4	4	4
B	4	5	3	8
B	7	8	4	10

Table 1: Node labels

Table 1 illustrates the node labels assigned to *A* and *B* nodes in Figure 1. We can see that the parent id of the third *A* and second *B* are equal because they are siblings.

Once these numbers are assigned to each node, the nodes can be stored independently without loss of any structural information (in either a relational database or an inverted index). At query execution time, the set of elements on either side of an operator are extracted and only those node numbers that satisfy the operator's specification are selected as the result. For example, if the operator is the child relation, and the operands are *A* and *B*, then there are two matches: $B\{3, 4, 4, 4\}$, child of $A\{2, 4, 3, 2\}$ and, $B\{7, 8, 4, 10\}$, child of $A\{5, 8, 3, 8\}$.[2] This process of finding the elements of a document that match operators is nothing other than the standard join operation (and it is implemented differently in relational databases and IR engines).

3.2 Relational database approach

Tree nodes can be stored in a relational database using a table structure (Bird et al., 2006). Each treebank would have a single table for all nodes where each node's information is stored in a tuple. The node name is stored along with other position information and the sentence id. Every node tuple also has a unique primary key. The parent id column is a foreign key, referencing the parent node's id, speeding up parent/child join operations. In practice, queries are translated from higher level linguistic query languages such as LPath into SQL automatically, allowing users to use a convenient syntax, rather than query using SQL.

Previous research on a similar database structure for containment queries in XML databases showed

[2]The node labels are represented as an ordered set here for brevity. Their positions match the headings in Table 1.

that databases are generally slower than specialised IR indexes (Zhang et al., 2001). In that work, the authors provide results comparing their IR join algorithm, the multi-predicate merge join (MPMGJN), with two standard relational join algorithms. They consider the number of comparisons performed in the standard merge join and the index nested loop join, and contrast these with their IR join algorithm. They show that the IR algorithm performs fewer comparisons than a standard merge join but greater than the index nested loop join.

The multi-predicate merge join exploits the fact that nodes are encountered in document order (i.e. a node appears before its descendents). Search within a document can be aborted as soon as it is clear that further searching will not yield further results. Importantly, this IR join algorithm is faster than both relational join algorithms in practice, since it makes much better use of the hardware disk cache. Our own experiments with a large treebank stored in an Oracle database have demonstrated that this shortcoming of relational query relative to IR query exists in the linguistic domain (Ghodke and Bird, 2008).

3.3 IR engine approach

We transform the task of searching treebanks into a conventional document retrieval task in which each sentence is treated as a document. Tree node labels are stored in inverted indexes just like words in a text index. We require two types of indexes, for frequency and position. The frequency index for a node label contains a list of sentence ids and, for each one, a count indicating the frequency of the node label in the sentence. (Labels with a frequency of zero do not appear in this index.) The position index is used to store node numbers for each occurrence of the node label. The numbers at each position are read into memory as objects only when required (at other times, the byte numbers are skipped over for efficiency). During query processing, the frequency indexes are first traversed sequentially to find a document that contains all the required elements in the query. Once a document is found, the structural constraints are checked using the data stored in the position index for that document. The document itself does not need to be loaded.

Using an inverted index for searching structured data is not new, and several XML databases already use this method to index XML elements (Meier, 2003). However, linguistic query systems are special purpose applications where the unit of retrieval is usually a sentence. A given tree may satisfy a query in multiple places, but we only identify which sentences are relevant. Finding all matches within a sentence requires further processing. [3]

Our approach has been to process each sentence as a document. By fixing the unit of retrieval to be the sentence, we are able to greatly reduce the size of intermediate results when performing a series of joins. The task is then to simply check whether a sentence satisfies a query or not. This can be done using substantially less resources than is needed for finding sets of nodes, the unit of retrieval for relational and XML databases. When processing a series of joins, we use a single buffer to store the node positions required to perform the next join in the series. After computing that join and processing another operator in the query, the buffer contents is replaced with a new set of nodes, discarding the intermediate information.

4 Experiments with IR Engine

4.1 Data

We used two data collections in our experiments. The first collection is a portion of the English Gigaword Corpus, parsed in the Penn Treebank format. We used the TnT tagger and the DBParser trained on the Wall Street Journal section of the Penn Treebank to parse sentences in the corpus. The total size of the corpus is about 5.5 million sentences. The TGrep2 corpus file for this corpus is about 1.8 GB and the Lucene index is 4 GB on disk. The second data collection is a portion of English Wikipedia, again tagged and parsed using TnT tagger and DBParser, respectively. This collection contains 26.5 million parsed sentences. The TGrep2 corpus file corresponding to this collection is about 6.6 GB and the Lucene index is 14 GB on disk.

[3]Several alternate path joins and improvements to the MPMGJN algorithm have been proposed over the years to overcome the problem of large number of intermediate nodes and to reduce unnecessary joins (Al-Khalifa et al., 2002; Li and Moon, 2001). Bruno *et al.*'s work on twig joins further improved on those efforts by processing an entire query twig in a holistic fashion (Bruno et al., 2002), and has since been further optimized.

Query	Selectivity	Data Collection 1 (5.5M sentences)					Data Collection 2 (26.5M sentences)				
		Full search			First 10		Full search			First 10	
(//N1 op N2)	N1-op-N2	cold	warm	hits	cold	warm	cold	warm	hits	cold	warm
NP/NN	L-L-L	7.326	5.533	4,814,540	0.059	0.0003	24.680	20.256	21,906,349	0.260	0.0003
VP/DT	L-H-L	4.576	3.593	17,328	0.140	0.004	13.865	11.363	91,070	0.301	0.003
NP/LST	L-L-H	4.454	0.043	6,808	0.083	0.001	16.864	0.077	2.974	0.270	0.003
VP/WHPP	L-H-H	2.445	0.034	32	1.012	0.014	8.834	0.066	29	3.653	0.015
LST\NP	H-L-L	4.444	0.043	6,808	0.080	0.001	16.814	0.077	2,974	0.271	0.003
WHPP\VP	H-H-L	2.461	0.034	32	0.990	0.013	8.726	0.065	29	3.611	0.015
LST/LS	H-L-H	0.181	0.005	10,432	0.071	0.0001	0.294	0.008	8,977	0.238	0.0002
LST/FW	H-H-H	0.123	0.009	4	0.103	0.011	0.348	0.012	9	0.408	0.012

Table 2: Execution times (in seconds) for queries of varying selectivity

4.2 Types of queries

Query performance depends largely on the nature of the individual queries, therefore we present a detailed analysis of the query types and their corresponding results in this section.

Selectivity: A query term that has few corresponding hits in the corpus will be considered to have high selectivity. The selectivity of whole queries depends not only on the selectivity of their individual elements, but also on how frequently these terms satisfy the structural constraints specified by the query.

Table 2 gives execution times for queries with varying selectivity, using our system. We assign a selectivity measure for the operator based on how often the two operands satisfy the structural condition. It is clear that when elements are very common and they frequently satisfy the structural constraints of the operator, there are bound to be more run-time structural checks and the performance deteriorates. This is demonstrated by the time taken by the first query. Note the relatively small difference in the execution time between the second and third queries. The third query contains a high selectivity element and even returns fewer matches compared to the second, but takes almost as long. This may be due to the relative frequency of the tags within each sentence, which we have not controled in this experiment. If there are several LST tags in the sentences where it appears, there are likely to be greater number of searches within each sentence. A better join algorithm would improve the performance in such cases.

A multiple regression analysis of the full search (cold start) times for collection 2 shows that low-selectivity labels contribute 9.5 seconds, and a low-selectivity operator contributes 6.7 seconds, and that this accounts for most of the variability in the timing data ($t = -1.53 + 9.51 * N_1 + 6.72 * op + 9.44 * N_2, R^2 = 0.8976$). This demonstrates that the distribution of full search (cold start) times is mostly accounted for by the index load time, with the time for computing a large join being a secondary cost. The full search (warm start) times in Table 2 pay a lesser index loading cost.

Query length: It is evident that the system must retrieve and process more term vectors as we increase the number of elements in a query. To find out exactly how the query length affects processing, we ran tests with three sets of queries. In each set we varied the number of elements in a dominance relationship with another node of the same name. The number of terms in the dominance relationship was varied from 1 to 6, where the first case is equivalent to just finding all terms with that name. In the first set, queries search for nested noun phrases (`NP`), while the second and third look for adjective phrases (`ADJP`) and list elements (`LST`) respectively.

These terms have been chosen to simultaneously study the effects of selectivity and query length, with `NP` being the least selective (or most common), followed by `ADJP`, then with `LST` being the most selective (or least common). `NP` is also more frequently self-nested than the others. Figure 2 plots query length (x-axis) against query execution time (y-axis, log scale) for the three sets, using our system. With

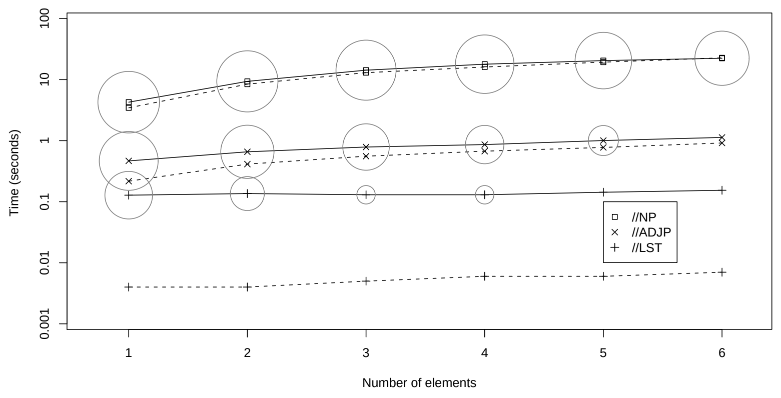

Figure 2: Variation of query execution time with query length in data collection 1

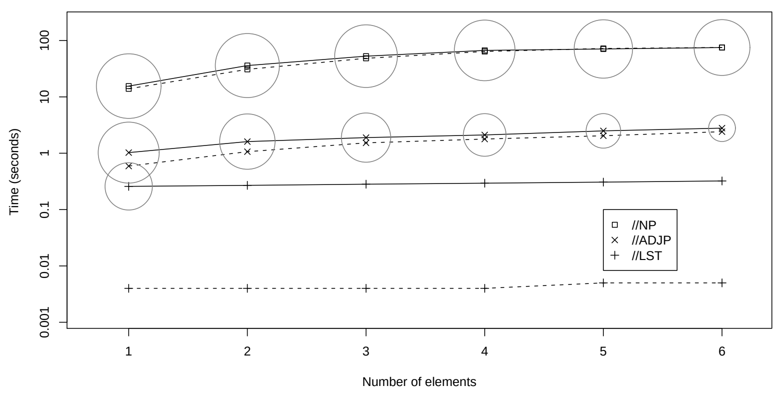

Figure 3: Variation of query execution time with query length in data collection 2

each step on the x-axis, a query will have an extra descendant node. For example, at position 3 for element `A`, the query would be `//A//A//A`.

The circles on the plot are proportional to the log of the result set size. The biggest circle is for `//NP` which is of the order of 5.4 million, while there are only 4 trees in which `LST` is nested 4 times. `LST` is not nested 5 or more times. Similarly, `ADJP` returns 0 results for the 6th test query and hence there are no circles at these locations. The thick lines on the plot indicate the average cold start run time over three runs, while the dashed line shows the minimum average run time of 4 sets, with the query executed 5 times in each set. Together, the pairs of unbroken and dashed lines indicate the variation in run time depending on the state of the system. [4]

4.3 Measurement techniques

The measurement techniques vary for TGrep and the IR based approach. In TGrep the corpus file is loaded each time during query processing, but in the IR approach an index once loaded can operate faster than a cold start.

In order to understand the variations in the operating speed we plot the variation in times from a cold start to a repeat query, as shown in Table 3.

[4]We can observe from the results that the variation between cold start and warm start correlates with query length. The length experiment here use a single term repeated multiple times. However, there is a possibility that the results may vary when the terms are different, because it would involve additional time to load the term vectors of distinct elements into memory.

Query	Data collection 1 TGrep2	Data collection 1 IR	Data collection 2 TGrep2	Data collection 2 IR
//NP	25.28	8.15	89.35	15.53
//NP//NP	25.44	10.42	88.36	35.95
//NP//NP//NP	25.45	14.96	87.48	52.81
//NP. . . //NP (4 times)	25.34	18.38	88.28	66.80
//NP. . . //NP (5 times)	25.46	20.94	87.38	70.80
//NP. . . //NP (6 times)	25.41	23.23	86.92	75.05
//ADJP	25.48	0.69	86.83	1.03
//ADJP//ADJP	25.36	0.73	86.42	1.61
//ADJP//ADJP//ADJP	25.29	0.84	86.89	1.89
//ADJP. . . //ADJP (4 times)	25.45	0.90	87.39	2.11
//ADJP. . . //ADJP (5 times)	25.23	1.03	86.50	2.49
//ADJP. . . //ADJP (6 times)	25.74	1.11	89.24	2.79
//LST	25.29	0.17	87.73	0.26
//LST//LST	25.49	0.20	87.09	0.27
//LST//LST//LST	25.38	0.20	87.66	0.28
//LST. . . //LST (4 times)	25.43	0.19	87.17	0.29
//LST. . . //LST (5 times)	25.40	0.19	88.02	0.31
//LST. . . //LST (6 times)	25.32	0.19	89.01	0.32
//NP/NN	25.66	7.33	87.63	24.68
//VP/DT	25.53	4.58	89.85	13.86
//NP/LST	25.62	4.45	86.39	16.86
//VP/WHPP	25.09	2.97	87.43	8.83
//WHPP/IN	25.75	4.44	88.48	16.81
//LST/JJ	25.46	2.46	86.57	8.73
//LST/LS	25.38	0.18	87.40	0.29
//LST/FW	25.51	0.12	87.27	0.35

Table 3: Comparison of TGrep2 and IR Engine cold start query times (seconds)

5 Conclusions

We have shown how an IR engine can be used to build a high performance tree query system. It outperforms existing approaches using indexless in-memory search, or custom indexes, or relational database systems, or XML database systems. We reported the results of a variety of experiments to demonstrate the efficiency of query for a variety of query types on two treebanks consisting of around 5 and 26 million sentences, more than two orders of magnitude larger than what existing systems support. The approach is quite general, and not limited to particular treebank formats or query languages. This work suggests that web-scale tree query may soon be feasible. This opens the door to some interesting possibilities: augmenting web search with syntactic constraints, the ability discover rare examples of particular syntactic constructions, and as a technique for garnering better statistics and more sensitive features for the purpose of constructing language models.

Acknowledgments

We gratefully acknowledge support from Microsoft Research India and the University of Melbourne.

References

Anne Abeillé, editor. 2003. *Treebanks: Building and Using Parsed Corpora*. Text, Speech and Language Technology. Kluwer.

Shurug Al-Khalifa, H.V. Jagadish, Nick Koudas, Jignesh M. Patel, Divesh Srivastava, and Yuqing Wu. 2002. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE '02: Proc. 18th Intl Conf on Data Engineering*, page 141. IEEE Computer Society.

Steven Bird, Yi Chen, Susan B. Davidson, Haejoong Lee, and Yifeng Zheng. 2006. Designing and evaluating an XPath dialect for linguistic queries. In *ICDE '06: Proc. 22nd Intl Conf on Data Engineering*, page 52. IEEE Computer Society.

Nicolas Bruno, Nick Koudas, and Divesh Srivastava. 2002. Holistic twig joins: optimal XML pattern matching. In *SIGMOD '02: Proc. 2002 ACM SIGMOD Intl Conf on Management of Data*, pages 310–321. ACM.

Steve Cassidy and Jonathan Harrington. 2001. Multi-level annotation of speech: an overview of the Emu Speech Database Management System. *Speech Communication*, 33:61–77.

Steve Cassidy. 2002. Xquery as an annotation query language: a use case analysis. In *Proc. 3rd LREC*.

John Chen and Owen Rambow. 2003. Use of deep linguistic features for the recognition and labeling of semantic arguments. In *Empirical Methods in Natural Language Processing*, pages 41–48.

Abdur Chowdhury and M. Catherine McCabe. 1998. Performance improvements to vector space information retrieval systems with POS. U Maryland.

Stephen Clark and James R. Curran. 2007. Wide-coverage efficient statistical parsing with CCG and log-linear models. *Computational Linguistics*, 33(4):493–552.

Michael Collins, Brian Roark, and Murat Saraclar. 2005. Discriminative syntactic language modeling for speech recognition. In *Proc. 43rd ACL*, pages 507–514. ACL.

Sumukh Ghodke and Steven Bird. 2008. Querying linguistic annotations. In *Proc. 13th Australasian Document Computing Symposium*, pages 69–72.

Jörg Hakenberg, Illes Solt, Domonkos Tikk, Luis Tari, Astrid Rheinländer, Nguyen Quang Long, Graciela Gonzalez, and Ulf Leser. 2009. Molecular event extraction from Link Grammar parse trees. In *Proc. BioNLP 2009 Workshop*, pages 86–94. ACL.

Philip J Harding, Quanzhong Li, and Bongki Moon. 2003. XISS/R: XML indexing and storage system using RDBMS. In *Proc. 29th Intl Conf on Very Large Data Bases*, pages 1073–1076. Morgan Kaufmann.

Ulrich Heid, Holger Voormann, Jan-Torsten Milde, Ulrike Gut, Katrin Erk, and Sebastian Pado. 2004. Querying both time-aligned and hierarchical corpora with NXT search. In *Proc. 4th LREC*.

Ichikawa Hiroshi, Noguchi Masaki, Hashimoto Taiichi, Tokunaga Takenobu, and Tanaka Hozumi. 2005. eBonsai: An integrated environment for annotating treebanks. In *Proc. 2nd IJCNLP*, pages 108–113.

Julia Hockenmaier and Mark Steedman. 2007. CCGbank: A corpus of CCG derivations and dependency structures extracted from the Penn Treebank. *Computational Linguistics*, 33:355–396.

Hiroshi Ichikawa, Keita Hakoda, Taiichi Hashimoto, and Takenobu Tokunaga. 2006. Efficient sentence retrieval based on syntactic structure. In *COLING/ACL*, pages 399–406.

Stephan Kepser. 2003. Finite Structure Query: A tool for querying syntactically annotated corpora. In *Proc. 10th EACL*, pages 179–186.

Esther König and Wolfgang Lezius. 2001. The TIGER language: a description language for syntax graphs. part 1: User's guidelines. Technical report, University of Stuttgart.

Catherine Lai and Steven Bird. 2004. Querying and updating treebanks: A critical survey and requirements analysis. In *Proc. Australasian Language Technology Workshop*, pages 139–146.

Catherine Lai and Steven Bird. 2010. Querying linguistic trees. *Journal of Logic, Language and Information*, 19:53–73.

Quanzhong Li and Bongki Moon. 2001. Indexing and querying XML data for regular path expressions. In *VLDB '01: Proc. 27th Intl Conf on Very Large Data Bases*, pages 361–370. Morgan Kaufmann.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–30.

Hendrik Maryns and Stephan Kepser. 2009. Monasearch: Querying linguistic treebanks with monadic second-order logic. In *The 7th International Workshop on Treebanks and Linguistic Theories*.

Neil Mayo, Jonathan Kilgour, and Jean Carletta. 2006. Towards an alternative implementation of nxts query language via xquery. In *Proc. 5th Workshop on NLP and XML: Multi-Dimensional Markup in Natural Language Processing*, pages 27–34. ACL.

J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. 1997. Lore: A database management system for semistructured data. *SIGMOD Rec.*, 26:54–66.

Wolfgang Meier. 2003. eXist: An open source native XML database. In *Revised Papers from the NODe 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems*, pages 169–183. Springer-Verlag.

Jiří Mírovský. 2006. Netgraph: a tool for searching in Prague Dependency Treebank 2.0. In *Proc. 5th Intl Conf on Treebanks and Linguistic Theories*, pages 211–222.

Jiří Mírovský. 2008. PDT 2.0 requirements on a query language. In *Proc. 46th ACL*, pages 37–45. ACL.

Preslav Nakov, Ariel Schwartz, Brian Wolf, and Marti Hearst. 2005. Supporting annotation layers for natural language processing. In *Proc. 43rd ACL*, pages 65–68.

Stephan Oepen, Kristina Toutanova, Stuart Shieber, Christopher Manning, Dan Flickinger, and Thorsten Brants. 2002. The LinGO Redwoods Treebank: Motivation and preliminary applications. In *Proc. 19th COLING*, pages 1253–57.

Petr Pajas and Jan Štěpánek. 2009. System for querying syntactically annotated corpora. In *Proc. 47th ACL*, pages 33–36. ACL.

Douglas L. T. Rohde, 2005. *TGrep2 User Manual Version 1.15*. http://tedlab.mit.edu/ dr/TGrep2/tgrep2.pdf.

Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. 2002. Storing and querying ordered XML using a relational database system. In *SIGMOD '02: Proc. 2002 ACM SIGMOD Intl Conf on Management of Data*, pages 204–215. ACM.

M. Čmejrek, J. Cuřín, and J. Havelka. 2004. Prague czech-english dependency treebank: Any hopes for a common annotation scheme? In A. Meyers, editor, *HLT-NAACL 2004 Workshop: Frontiers in Corpus Annotation*, pages 47–54. ACL.

Martin Volk, Joakim Lundborg, and Maël Mettler. 2007. A search tool for parallel treebanks. In *Proc. Linguistic Annotation Workshop*, pages 85–92. ACL.

Sean Wallis. 2003. Completing parsed corpora. In Anne Abeillé, editor, *Treebanks: Building and Using Parsed Corpora*, Text, Speech and Language Technology, pages 61–71. Kluwer.

Naiwen Xue, Fei Xia, Fu-Dong Chiou, and Martha Palmer. 2005. The Penn Chinese TreeBank: Phrase structure annotation of a large corpus. *Natural Language Engineering*, 11:207–238.

Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. 2001. On supporting containment queries in relational database management systems. In *SIGMOD '01: Proc. ACM SIGMOD international Conference on Management of Data*, pages 425–436, New York. ACM.