

## Accelerating Corpus Search Using Multiple Cores

Radoslav Rábara, Pavel Rychlý, Ondřej Herman and Miloš Jakubíček

Masaryk University

Botanická 68a

Brno, Czech Republic

{xrabara, pary, xherman1, jak}@fi.muni.cz

### Abstract

The Manatee corpus management system on which the Sketch Engine is built is efficient, but unable to harness the power of today's multiprocessor machines. We describe a new, compatible implementation of Manatee which we develop in the Go language and report on the performance gains that we obtained.

### 1 Introduction

Text corpora are huge collections of texts in electronic form. They are used as an empirical resource for observation of real world language use, to study the behavior of words, their meanings and the contexts they occur in. Corpora are employed in many fields of linguistics (morphology, syntax, semantics, stylistics, sociolinguistics etc.) Important tools enabling corpus exploration are corpus managers. Corpus managers have to be able to deal with extremely large corpora effectively and provide platform for complex query evaluation, result filtering and visualization and computation of a wide range of lexical statistics. Processing speed is an important aspect of their operation because of the size of the corpora – billions of words and more. In order to speed up the processing, we reimplemented the single-threaded query evaluation engine in a concurrent way within the Manatee corpus management system (Rychlý, 2007).

### 2 Manatee system

The Manatee system (Rychlý, 2000) is a corpus manager, designed to be able to deal with extremely large corpora, optimized for fast query evaluation. It consists of an indexing library for text compression, index building and search, a query evaluation module, a query parser which transforms the textual query representation into abstract syntax trees, a set of command line tools

for corpus building and maintenance and a graphical user interface. The system is based on the text indexing library FinLib which provides procedures for word indexing, corpus storage and retrieval of words in form of streams of positions (Rychlý, 2000). Manatee has its own query language, CQL, which enables users to execute complex queries on the corpus text.

The implementation of the query evaluation engine within Manatee is based on streams of tokens or token pairs, representing ranges or spans of consecutive tokens. The C++ `FastStream` and `RangeStream` interfaces represent token and range streams. Classes which implement them represent specific operations. The main idea is to have classes that perform simple operations which can be combined together to perform complex operations. In the original implementation, these classes are based on the iterator pattern. This means that only one value from the stream is available at any given time. The next value is loaded by calling the `next` method. Once it is called, the previous values are no longer available. Values are always provided in increasing order. After all values are read, an iterator returns a sentinel value, which is different from any other value in the stream. The iterator also provides a `find` method to seek to the next interesting value in an efficient way and a `peek` method to get the following value, which will be returned by calling the `next` method, without proceeding to the next position. More details about the implementation are in (Rychlý, 2000).

### 3 The Go programming language

Go, also referred to as Golang, is a new programming language which is being developed since 2007 by Google. Go tries to combine performance and security advantages of compiled language like C++ with the development speed of a dynamic language like Python (Pike, 2012). The language pro-

vides language-level parallelism through the so-called goroutines. A goroutine is a coroutine attached to a thread. Multiple coroutines can share a single thread to conserve operating system resources. The attachment is performed dynamically by the Go runtime. Communication and synchronization between goroutines is carried out using channels. Channels are used to move data from a sender to a receiver. The communication blocks the sender until the receiver receives the message. In this way, channels can provide synchronization between goroutines without explicit locks or condition variables. Channels can also be buffered. Buffered channel operations block the sender only when the buffer is full and they block the receiver only when the buffer is empty. More details about Go channels can be found in (Pike et al., 2012). The new implementation of Manatee is being developed in Go.

## 4 Implementation

In the new implementation, `FastStream` and `RangeStream` have been modified to provide a channel as the stream of the positions in place of the iteration protocol. The original methods `next`, `peek`, and `find` are no longer provided because their functionality is provided by the channel itself. The `find` operation was removed from the new implementation as it was, surprisingly, slowing the application down. In most cases we expected the `find` operation to improve the application performance by reducing the amount of the data transferred.

Not everything that the old implementation supports has been reimplemented and conversely, some functionality is not present in the old system, but the core functionalities of the old and the new systems are very similar.

We compared the complexity of a few selected modules which provide the same functionality by counting the lines of source code that are not blank or comment lines. Of the 4 compared modules, 3 of them (`FastStream` which represents sequences of positions, `Read_bits` and `Write_bits` which provide access to the compressed data storage, `SortedRuns` employed in lexicon construction) have nearly the same length in both of the implementations, the other module, `RangeStream`, used for handling sequences of structures or spans of text, is actually 30% shorter in the new implementation, even

though it supports concurrent query evaluation and employs some boilerplate to speed up communication over Go channels by sending larger batches.

## 5 Performance comparison

The evaluation of both of the implementations was performed on an eight core server. The original implementation is a single-thread application, so it is able to use only a single processor core. The new implementation, which has been designed in a concurrent way with goroutines, can exploit multiple cores. The new implementation was evaluated with different number of threads enabled. The performance was compared using a benchmark which measured the time of evaluation of a set of prepared queries. The prepared evaluation queries are complex and difficult to evaluate as they cover rules of syntactic analysis. The result of each of the queries is big, it covers approximately 5% to 10% of the whole corpus. The combined results of all the queries cover almost the whole corpus. These queries are quite extreme, but allow for more thorough evaluation of the system performance. Typical users of the Manatee system usually create simple queries to find specific words with a few restrictions, which usually produce small result sets.

The original implementation evaluated the prepared queries in 4h 29m 24s. The new implementation evaluated the queries in 2h 27m 39s, when running only on a single thread. Compared to the original implementation, the new implementation is faster by approximately 45%. The results show that 13 of 15 of the queries were evaluated faster by the new implementation with an average speedup of approximately 45%. The best improvement for a single query is 82%. The smallest improvement is 20%. Only two of the queries were evaluated slower by the new implementation. One was evaluated slower by 116% and the second one was evaluated slower by 7%.

The new implementation was evaluated with different amount of threads enabled, so that we could observe the performance scaling. As shown in Figure 1, the evaluation with three threads sped up the evaluation by 32% compared to the evaluation with two threads and by 133% compared to the evaluation with one thread. Four threads speed up the evaluation by 23% compared to the evaluation with three threads and by 186% comparing to the evaluation with one thread. Adding the fifth

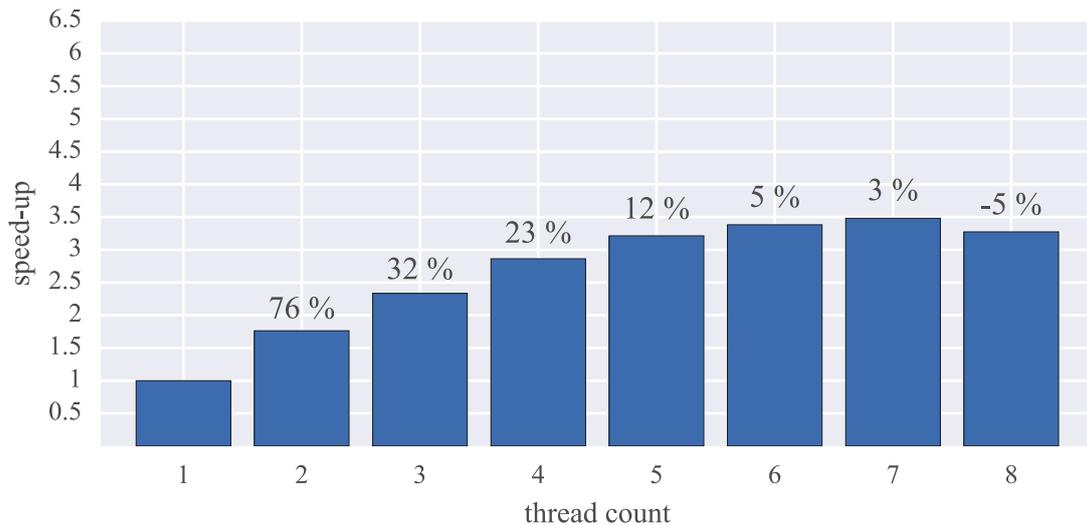


Figure 1: Average scaling over the whole testsuite

thread increases the performance by 12% compared to the evaluation with four threads. Runs with six and seven threads differ in less than 5% compared to runs with one thread. Using eight threads is actually slower than seven threads, likely due to I/O contention and cache spills.

## 6 Discussion

The speed-up observed with additional threads is more pronounced for complex queries, while simple queries might not scale at all. For example, a simple query of the form `[word="at"]` does not show any improvement in runtime when more threads are enabled, as can be seen in Figure 2. This is caused by the lack of the need for processing of the result. Most of the time needed to evaluate simple queries is spent on waiting for I/O and constructing the concordance.

A more complex query of the form `[word="at"] [tag="NN"]` benefits from up to three threads. The query engine evaluates simultaneously the positions of the token *at* and positions of nouns. Another process then combines these two streams of positions and picks the pairs which represent positions that are next to each other.

The query<sup>1</sup> used in calculation of Word

<sup>1</sup>`[word="it"] [tag="RB.?" | tag="RB" | tag="VM"]{0,3} [lemma="be" & tag="V.*"]? [tag="RB.?" ]{0,2} [tag="DT.?" | tag="PP$"]{0,1} [tag="CD"]{0,2} [tag="JJ.?" | tag="RB.?"`

Sketches (Kilgariff et al., 2001) scales almost linearly when additional threads are employed, as can be seen in Figure 4. This is because the evaluation of the query needs to combine many different sources of data. I/O throughput is still important, but most of the time is spent in processes which manipulate the and combine the streams coming from storage.

## 7 Future work

While the new implementation of the query evaluation system is not significantly faster for simple queries, it provides large speed-ups for evaluating complex queries which are used for the calculation of Word Sketches, terminology extraction and other advanced features of Sketch Engine.

The new implementation is already used for calculation of some performance-intensive tasks, such as for the calculation of the Longest-commonest match (Kilgarriff et al., 2015), which was nearly infeasible with the old implementation.

Most importantly, the new architecture lays the groundwork for distributing the query processing over a larger cluster of machines, where every machine operates on a small part of the corpus only. This will allow us to provide further performance improvements, avoid I/O bottlenecks and also improve our ability to provide more redundant and fault tolerant system.

`| word=", "]{0,3} [tag="N.*"]{0,2} 1:[tag="N.*"] [word="for"] [tag="PP"] [tag="TO"]`

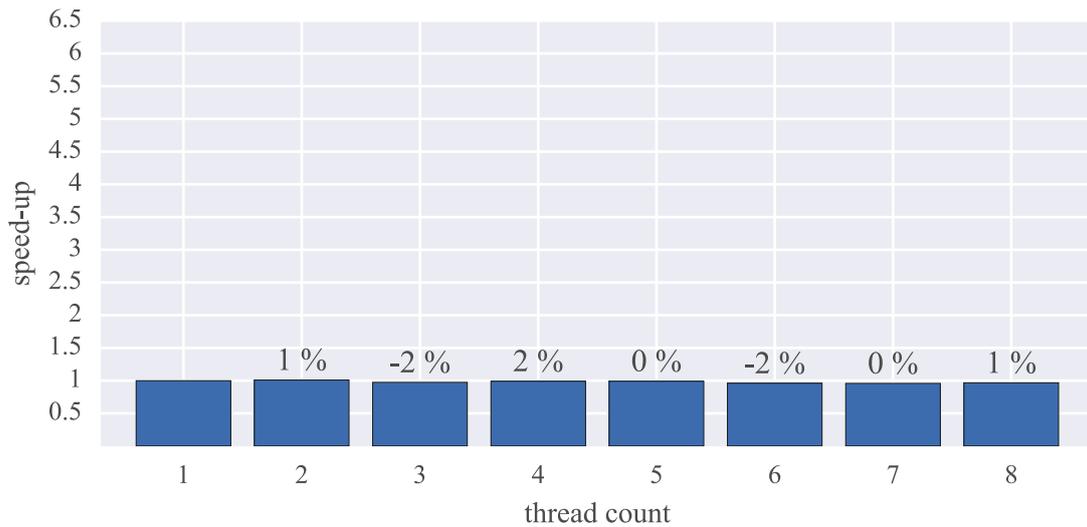


Figure 2: Scaling for a primitive query matching the word *at*

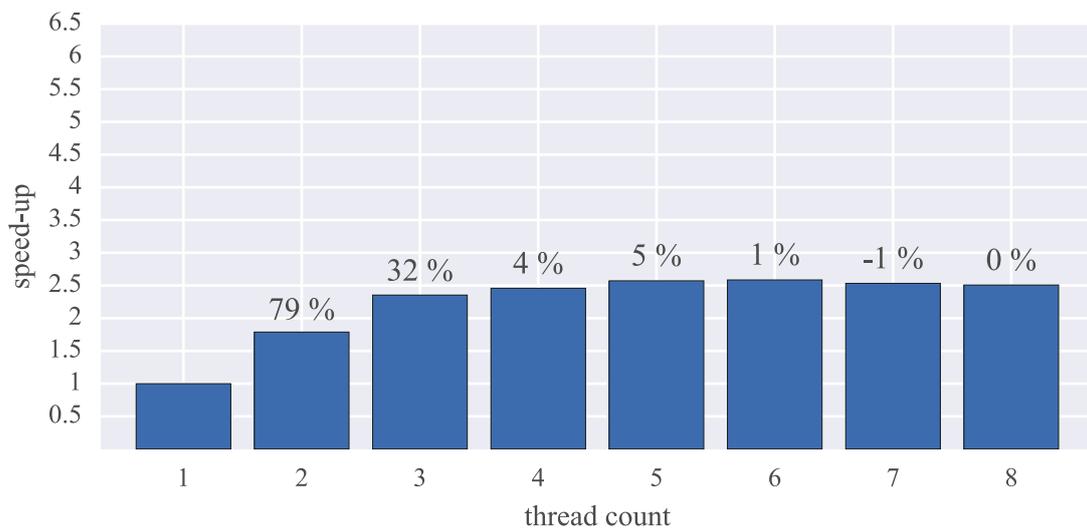


Figure 3: Scaling for a compound query matching sequences composed of *at* followed by a noun

## 8 Conclusion

The performance and the length of the source code were compared between the single-thread and concurrent implementation of the corpus manager Manatee. Manatee is able to deal with extremely large corpora and provides a platform for evaluating complex queries, filtering and visualizing results, and computing a wide range of lexical statistics (Kilgarriff et al., 2014). The original C++ implementation evaluated the prepared benchmark queries in approximately 4.5 hours. The new Go implementation managed to evalu-

ate the prepared benchmark queries on a single thread in approximately 2.5 hours. The concurrent system performed better by 45% on a single thread. The new implementation was also evaluated with different amount of enabled threads. The performance increased by 15.7% on average with each additional enabled thread of the server and the most significant enhancement by 76% was between running on one and two threads.

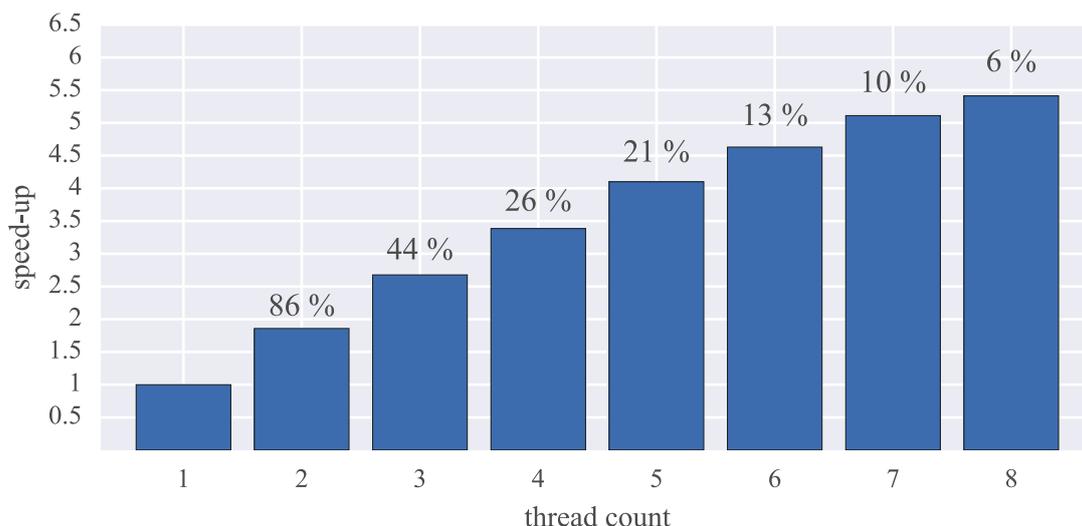


Figure 4: Scaling for a complex query matching phrases similar to *it's time for you to* and *it's not an intense thing for him to*

## Acknowledgments

Reimplementation of Manatee in the Go language was partially supported by Lexical Computing.

## References

- Pike, R. 2012. Go at Google: Language Design in the Service of Software Engineering. online, URL: <https://talks.golang.org/2012/splash.article>
- Pike, R., Gerrand, A. 2012. Concurrency is not parallelism. online, URL: <http://talks.golang.org/2012/waza.slide> Heroku Waza.
- Rychlý, P. 2000. Corpus Managers and their effective implementation. PhD Thesis, Masaryk University.
- Rychlý, P. 2007. Manatee/Bonito - A Modular Corpus Manager. *Proceedings of the 1st Workshop on Recent Advances in Slavonic Natural Language Processing*. pp.65–70. Masaryk University, Brno.
- Kilgarriff, A., Baisa, V., Bušta, J., Jakubíček, M., Kovář, V., Michelfeit, J., Rychlý, P. and Suchomel, V. Lexicography, 1(1), pp.7-36. 2014. The Sketch Engine: ten years on. *Journal of the Association for Computing Machinery*, 28(1):114–133.
- Kilgarriff, A., Baisa, V., Rychlý, P. and Jakubíček, M. 2015. Longest–commonest Match. In *Electronic lexicography in the 21st century: linking lexical data in the digital age. Proceedings of the eLex 2015 conference* (pp. 11-13).
- Kilgarriff, A., Tugwell, D. 2001. Word sketch: Extraction and display of significant collocations for lexicography.