MASARYK UNIVERSITY
FACULTY OF INFORMATICS

Parallelization of the corpus manager's time-consuming operations

MASTER'S THESIS

Bc. Radoslav Rábara

Brno, 2016

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Advisor: doc. Mgr. Pavel Rychlý, Ph.D.

Acknowledgement

First of all, I would like to thank my advisor doc. Mgr. Pavel Rychlý, Ph.D. for all his patience, advices, and guidance in the last four years. It was a great opportunity for me to learn from excellent teacher and programmer. I would also like to thank my family, girlfriend, and friends for supporting me.

Abstract

The Manatee corpus manager can process large corpora containing billions of words. Some operations with search results from such large corpora can be time-consuming. This thesis provides and describes a system that enables computation of the selected operations in parallel. The system is evaluated on a single computer, and on a cluster of computers. The evaluation contains evaluation of the scalability, and comparions with the Manatee system and a MapReduce system that provides a platform for distributed computing.

Keywords

Manatee, text corpora, Go, text compression, compression of integers, time-consuming operations, parallelization, MapReduce, Glow, distributed computing, data format, cluster, scalability

Contents

Chapter 1

Introduction

Text corpora are collections of text in electronic form [25]. As kind of empirical data, they are used in linguistics and related disciplines [2]. Thanks to the Internet and web crawling, it is relatively easy to create extreme large corpora containing billions of words and providing more information that can be used in a research [20].

Large corpora can be accessible by modern corpus manager systems, but some operations can be time-consuming. Time-consuming operations create unpleasant user experience and consume hardware resources, so all other operations are also slowed down. This thesis presents and describes a solution enhancing the performance of corpus manager's operations by their parallelization on a cluster of computers.

1.1 Manatee

Manatee is a state-of-the-art corpus manager system [9]. It is capable of managing even extremely large corpora with billions of words [20]. The basic function is retrieving results for a given query from a specified corpus. A corpus is queried by an extended Corpus Query Language (CQL, [10]). List of all occurrences for a given query is called concordance [25]. Concordance can be used to perform additional operations like sorting or computing a frequency distribution.

Bonito is a web-based graphical user interface of the Manatee system [26]. It displays a concordance in form of KWIC (keywords in context) lines. The KWIC lines are based on KWIC format, which shows one line for each occurrence of a concordance [25]. Bonito lays out the KWIC lines to separate pages and displays one page at a time. This approach enables a parallel computation of a concordance in a background, while a user can already see the first page of results.

Some operations can produce the results only after processing of the whole concordance. The concordance can be every word from a corpus, so if the corpus is large, the operations can be time-consuming as they process huge amount of data and often results in large number of disk reads.

As a concordance can contain any words from a corpus, it requires random access and makes impossible to optimize access to the memory by storing related data next to each other. Manatee optimizes access to the memory by creating and using indices. Another enhancement can be achieved by a technology of distributing data to mutliple disks, e.g. RAID 6. Data can be also distributed to a cluster of computers that can also

enable parallel computing.

1.2 Corpus Query Language

Corpus Query Language (CQL) is used to search for a sequence of words in the text corpora. The language was developed at University of Stuttgart in the early 1990s. Manatee uses the CQL and extends it in several ways. [13] This chapter provides brief description sufficient enough to underestand queries stated in this thesis.

1.2.1 Simple Attribute-Value Queries

Words in corpora can have attached multiple attributes, e.g. part-of-speech or syntactic categories. Every defined attribute can be queried. The basic attribute is the word attribute, which represent words of a text corpus.

Simple form to query a value of an attribute is **`[attribute="value"]`**, where the value can be a regular expression. For example, to find all occurrences of the words that has prefix **`confus`**, the query **`[word="confus.*"]`** can be used. Figure 1.1 shows the first 20 results of the query **`[word="test"]`** evaluated on the British National Corpus (BNC) [1].

The form can use two types of the comparison operators:

- operators evaluating the specified value as a regular expression: matches the regular expression **`=`**, or does not match the regular expression **`!=`**
- operators evaluating the specified value as a plain string: exact match **`==`**, inequality **`!==`**, less than or equal **`<=`**, greater than or equal **`>=`**, greater than **`!<=`** and less than **`!>=`**

To find a sequence of words, the query is expressed as a sequence of the related expressions. For example, the query to find all words **`blue`** followed by the word **`car`** can be expressed as **`[word="blue"] [word="car"]`**.

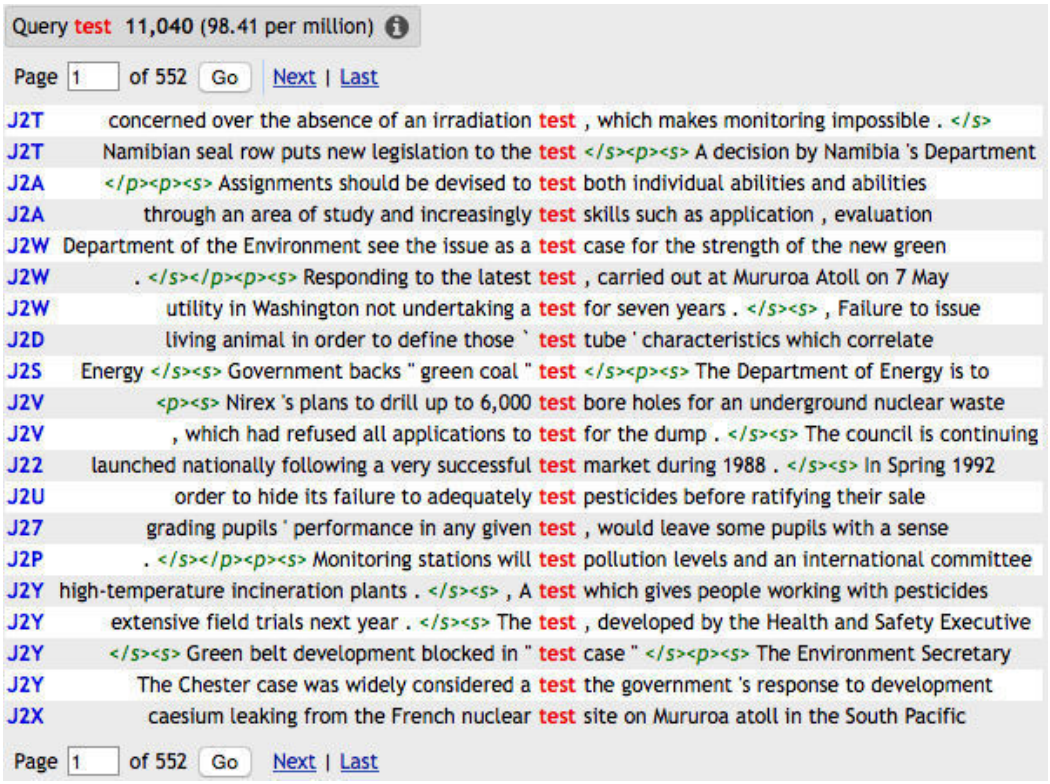
1.2.2 More Complex Expressions

The attribute-value expressions can be more complex. They can restrict or enlarge the suitable results. The restriction is denoted with the conjunction (**`&`**) and enlarging is denoted with the disjunction (**`|`**). For example, **`[word=".*ing" & tag="V.*"]`** finds all words that match the regular expression **`".*ing"`** and have the tag attribute matching the regular expression **`"V.*"`**.

More information about the query language can be found at [13].

1.3 Frequency Distribution and Sort of a Concordance

Frequency distribution computes a list of unique words and their number of occurrences. Sort of a concordance, referred to as concordance sort operation, sorts a con-



Query test 11,040 (98.41 per million)

Page 1 of 552 Go Next | Last

J2T	concerned over the absence of an irradiation test , which makes monitoring impossible . </s>
J2T	Namibian seal row puts new legislation to the test </s> <s> A decision by Namibia 's Department
J2A	<s> Assignments should be devised to test both individual abilities and abilities
J2A	through an area of study and increasingly test skills such as application , evaluation
J2W	Department of the Environment see the issue as a test case for the strength of the new green
J2W	. </s> <s> Responding to the latest test , carried out at Mururoa Atoll on 7 May
J2W	utility in Washington not undertaking a test for seven years . </s><s> , Failure to issue
J2D	living animal in order to define those ` test tube ' characteristics which correlate
J2S	Energy </s><s> Government backs " green coal " test </s> <s> The Department of Energy is to
J2V	<s> Nirex 's plans to drill up to 6,000 test bore holes for an underground nuclear waste
J2V	, which had refused all applications to test for the dump . </s><s> The council is continuing
J22	launched nationally following a very successful test market during 1988 . </s><s> In Spring 1992
J2U	order to hide its failure to adequately test pesticides before ratifying their sale
J27	grading pupils ' performance in any given test , would leave some pupils with a sense
J2P	. </s> <s> Monitoring stations will test pollution levels and an international committee
J2Y	high-temperature incineration plants . </s><s> , A test which gives people working with pesticides
J2Y	extensive field trials next year . </s><s> The test , developed by the Health and Safety Executive
J2Y	</s><s> Green belt development blocked in " test case " </s> <s> The Environment Secretary
J2Y	The Chester case was widely considered a test the government 's response to development
J2X	caesium leaking from the French nuclear test site on Mururoa atoll in the South Pacific

Page 1 of 552 Go Next | Last

Figure 1.1: The first 20 results of the query **`[word="test"]`** evaluated on the BNC corpus.

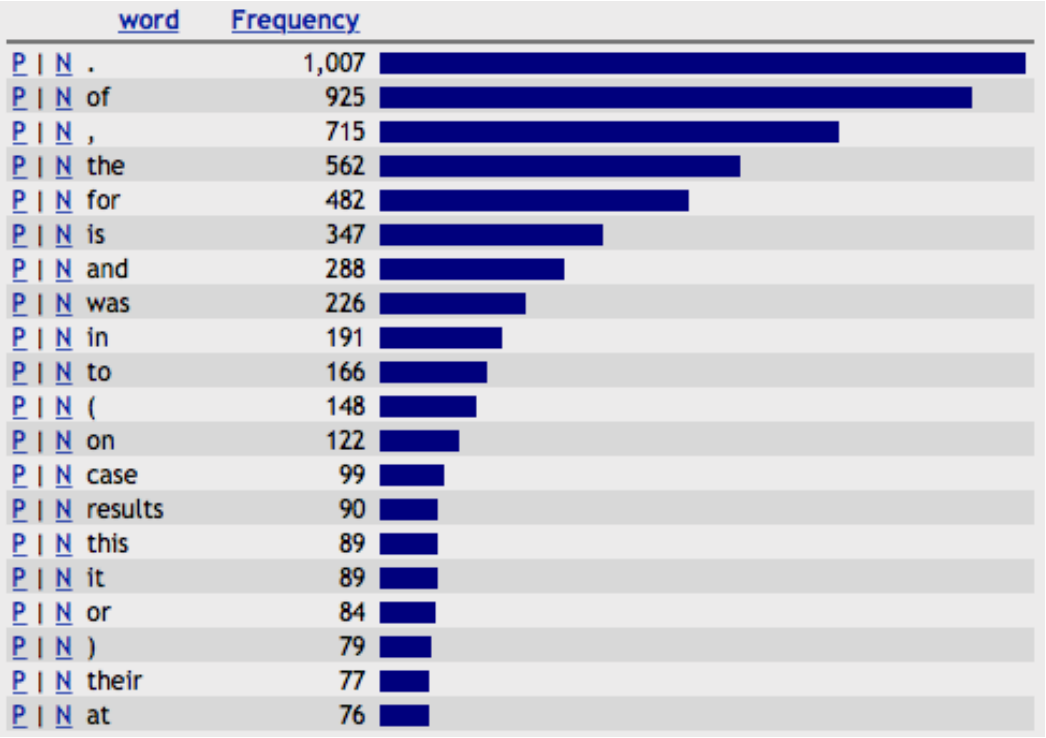
cordance. Both operations require a concordance and criteria. The given criteria are evaluated for each occurrence (item of a concordance), and the result is a line of text for each occurrence. Both operations process the lines. The frequency distribution computes number of occurrences for each unique line, and the concordance sort operation sorts the results and then sorts the concordance by the sorted results.

The result of frequency distribution is a list of pairs of the evaluated criteria and number of its occurences. Figure 1.2 shows the frequency distribution of a concordance of all occurrences of the word **`test`** and criterion returning the first word after an evaluated item. The operation is evaluated on the BNC corpus.

The concordance sort operation produces a concordance sorted by the evaluated criteria. Figure 1.3 shows the last 20 results of a sorted concordance of all occurrences of the word **`test`**. The concordance and operation are evaluated on the BNC corpus. The criterion of the concordance sort operation defines the right context of each concordance item as the following 3 words and it also converts the results to lower case, so the sort is case insensitive.

The operations have to process the whole concordance before any results are produced. The time to perform such operations depends on the size of the concordance. The size of a concordance is limited by a size of an appropriate corpus, so the concor-

dance can be huge and the operations time-consuming.



word	Frequency
P | N .	1,007
P | N of	925
P | N ,	715
P | N the	562
P | N for	482
P | N is	347
P | N and	288
P | N was	226
P | N in	191
P | N to	166
P | N (	148
P | N on	122
P | N case	99
P | N results	90
P | N this	89
P | N it	89
P | N or	84
P | N)	79
P | N their	77
P | N at	76

Figure 1.2: The first 20 results of the frequency distribution of the first words after the words **`test`** evaluated on the BNC corpus.

Criteria

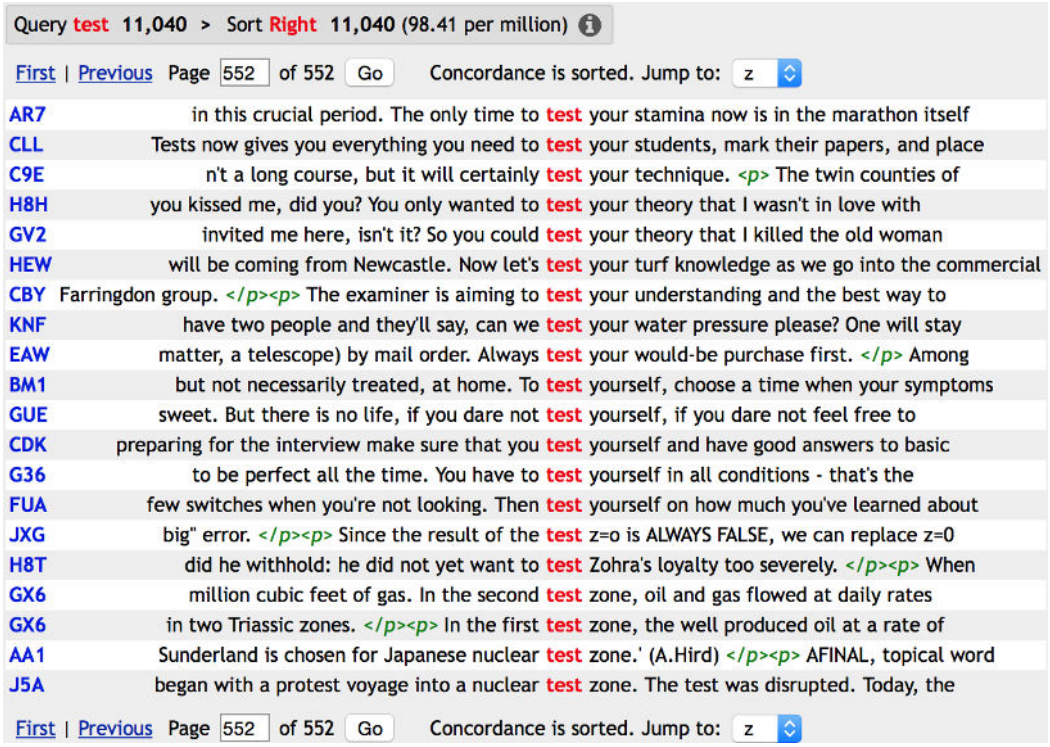
Criteria are quite complex as they have many options and variants. Each criterion defines attribute and context. Context can be node or collocation, or can define a range relative to the node or collocation. Node denotes the words matched by a query [14]. Collocation is a word of a node that is explicitly labeled in the query.

The following patterns and examples should help a reader to understand the criteria mentioned in this thesis.

Simple criteria can define the right or left context as one of the following or preceding words:

- **`word 1`** – defines the right context as the following word (the first word on the right side of a node).
- **`word 2`** – defines the right context as the second word on the right side of a node.
- **`word -2`** – defines the left context as the second word on the left side of a node.

The context can be defined from the beginning or the end of a node. It is useful when the node has variable number of words.



Query test 11,040 > Sort Right 11,040 (98.41 per million)

First | Previous Page 552 of 552 Go Concordance is sorted. Jump to: z

AR7	in this crucial period. The only time to test your stamina now is in the marathon itself
CLL	Tests now gives you everything you need to test your students, mark their papers, and place
C9E	n't a long course, but it will certainly test your technique. The twin counties of
H8H	you kissed me, did you? You only wanted to test your theory that I wasn't in love with
GV2	invited me here, isn't it? So you could test your theory that I killed the old woman
HEW	will be coming from Newcastle. Now let's test your turf knowledge as we go into the commercial
CBY	Farringdon group. The examiner is aiming to test your understanding and the best way to
KNF	have two people and they'll say, can we test your water pressure please? One will stay
EAW	matter, a telescope) by mail order. Always test your would-be purchase first. Among
BM1	but not necessarily treated, at home. To test yourself, choose a time when your symptoms
GUE	sweet. But there is no life, if you dare not test yourself, if you dare not feel free to
CDK	preparing for the interview make sure that you test yourself and have good answers to basic
G36	to be perfect all the time. You have to test yourself in all conditions - that's the
FUA	few switches when you're not looking. Then test yourself on how much you've learned about
JXG	big" error. Since the result of the test z=o is ALWAYS FALSE, we can replace z=0
H8T	did he withhold: he did not yet want to test Zohra's loyalty too severely. When
GX6	million cubic feet of gas. In the second test zone, oil and gas flowed at daily rates
GX6	in two Triassic zones. In the first test zone, the well produced oil at a rate of
AA1	Sunderland is chosen for Japanese nuclear test zone.' (A.Hird) AFINAL, topical word
J5A	began with a protest voyage into a nuclear test zone. The test was disrupted. Today, the

First | Previous Page 552 of 552 Go Concordance is sorted. Jump to: z

Figure 1.3: The last 20 results of a concordance for the query **`[word="test"]`** evaluated on the BNC corpus and sorted by the following 3 words with case insensitivity.

- **`word 1<0`** – defines the context as one word to the right from the beginning of the node. It is equivalent to **`word 1`**.
- **`word 1>0`** – defines the context as one word to the right from the end of the node. If, and only if, the node has only one word, the criterion is equivalent to **`word 1`**.

In the similar way, the context can refer to the beginning or the end of a collocation. A collocation is always part of a node and it is distinguished in the node by a numeric label. The numeric label is defined by an user in a given query and each collocation should have an unique number within a query.

- **`word 1<1`** – defines the context as one word to the right from the beginning of the first collocation (collocation with the label **`1`**).
- **`word 1>2`** – defines the context as one word to the right from the end of the second collocation (collocation with the label **`2`**).

The criteria can refer to more than just one word. They can define range of words.

- **`word 1~3`** – uses from the first to the third word on the right side as the context.

The notations can be combinged together to specify range of words from the beginning or the end of a node or collocation.

- **`word 1>0~3>1`** – defines the right context from the first word on the right side from the end of node, to the third word on the right side from the first collocation

The result of the criteria can be modified by predefined modificators. The modificators are listed after an attribute and between the attribute and the list is a separator (**`/`**).

- **`word/i 1`** – converts the results to a lower case, e.g. enables the concordance sort operation with case insensitivity.
- **`word/r 1`** – reverses the string results, e.g. enables sorting from the end of words.

1.4 Text Compression in Manatee

The Manatee system compresses text of a corpus in two steps: the first step is to map a string to an integer; the second step is encoding of the integer using Elias delta coding [6]. The mapping is provided by lexicons. The lexicon provides bidirectional mapping as each unique word has assigned a unique numeric identifier. The whole text of a corpus is encoded as a sequence of the numeric identifiers that can be converted to the words by using the lexicon.

Elias delta coding is a variable-size coding of integers. It does not support a random access to the encoded data, but each code can be decoded when its position in a bit stream is known. Manatee ensures relatively random access to any part of the encoded text by dividing the text into segments and storing positions of the segments. Each segment has the same size. It is usually 128 words for corpora with less than billion words and 64 words for corpora with more than billion words.

Table 1.1 shows files structure of the encoded text. Index of the lexicon ensures efficient mapping from an integer to string, and index of the sorted lexicon ensures efficient mapping from a string to integer. Not all files are required to dump the whole corpus. Such action requires only the encoded text, lexicon, and index of the lexicon.

File suffix	File description
.text	contains the encoded text – identifiers of the words encoded using Elias delta coding
.text.seg	index of the segments
.lex	unique string values in such order that their position is equal to the appropriate identifier
.lex.idx	index of the lexicon
.lex.srt	index of the sorted lexicon

Table 1.1: File structure of the text encoded by the Manatee system.

Chapter 2

Text Compression

Distribution of a corpus to a cluster of computers can result in inconsistency of the words's numeric identificators. Possible solution is to use the identificators only locally on a single computer, and the results propagate as the appropriate text. Although, the problem can be solved, it questions mapping of strings to integers and whether the current encoding should not be replaced.

There are many compression algorithms that are faster or have better compression ration than the text compression of the Manatee system. Manatee has to provide fast data access and efficient storage methods for even extremely large corpora with billions of words. Thus, the priorities are the compression ratio and decompression time. The compression time is not a priority because corpora are typically compiled once.

The new and better text compression should have faster decompression time and better compression ratio. It is also acceptable to have slightly worse compression ratio, if the decompression is faster. If the compression does not support the random access, the input text can be divided into segments and the segments can be compressed separately.

2.1 Text Compression Benchmarks

Manatee is an open-source software, so the selected algorithm has to be open source too. It is the reason why most of the evaluated programs are open-source. The benchmarks contains also a few closed-source programs (e.g. LzTurbo) as they declare better results. Although, the closed-source programs cannot be used to replace the Manatee's encoding, they can be at least compared.

The first benchmark evaluates 30 selected compression programs with different options. The benchmark contains 61 results in total. Many of the programs were selected by their results stated in Large Text Compression Benchmark [18], which ranks more than 190 text compression programs by the compression of enwik9 and enwik8. Enwik9 and enwik8 are dumps of the English Wikipedia version from 3. March 2006. Enwik9 contains the first 10^9 bytes and the enwik8 contains the first 10^8 bytes of the English Wikipedia. [17]

The second benchmark contains only 14 results of the best 10 compression algorithms, which were selected by their results in the first benchmark. The number of algorithms was limited, because the second benchmark is more complex and time-consuming.

Both benchmarks use data from the vertical text of the British National Corpus (BNC). BNC is a hundred million word text corpus of samples of written and spoken English [1]. The benchmarks were executed in the same environment: 64-bit Ubuntu 14.04.03 running as a virtual machine with 2 CPU cores and 2 GB RAM.

2.1.1 Evaluated Text Compression Programs

The evaluated compression programs use different algorithms. Most of them use or are derived from LZ77, e.g. LzTurbo uses LZ77 with arithmetic coding. Others use for example BWT, LZMA, LZMA2, LZP, byte-oriented LZP, LZSS followed by arithmetic coding, LZW, symbol ranking, or byte pair coding. More information about the programs can be found at [18].

LZ77 (also called "Sliding window") uses a sliding window of a fixed size divided into 2 parts: search buffer and look-ahead buffer. Search buffer contains already encoded text and is used as a dictionary for the current encoding. Look-ahead buffer contains text to be encoded. The first symbol in the look-ahead buffer is scanned in the search buffer from the right to the left, and if it is found, the encoder tries to match as many symbols as possible. The output contains triples consisting of an offset (distance) from the end of the search buffer, the length of the matched sequence, and the unmatched character. [27]

2.1.2 The First Text Compression Benchmark

The purpose of the first benchmark is to select some candidates that will be compared with the Manatee's text compression algorithm in the second benchmark. The second benchmark evaluates only the selected candidates – the best algorithms of the first benchmark.

The first benchmark uses a reduced vertical text of the BNC's word attribute, referred to as comparative text, to compare the algorithms. The comparative text consists of the first 28,202,868 lines (141 MB) of the BNC's word attribute, which has 112,181,015 lines (556 MB) in total. The Manatee system compresses the comparative text to 56 MB – 50 MB of the encoded text, 2.8 MB of the lexicon, and 1.2 MB of the lexicon's index. Other files, such as index of the sorted lexicon, are not relevant for decompression of the whole text. The comparative text was decompresed in 2.74 s.

All results of the first text compression benchmark are in Appendix A. The first 20 results of the first benchmark are shown in Table 2.1. The table is sorted by the decompression time and a table cell of the compressed file size column has gray background, when the size of the compressed file is greater than the size of the file compressed by Manatee (worse compression ratio than Manatee).

---

1. source: <https://github.com/Cyan4973/lz4> (last commit: d86dc91)
2. enabled multithreading
3. single-threaded
4. Unix command
5. multithreaded compression and single-threaded decompression

Program name	Options	Compression time [s]	Decompression time [s]	Compressed file size [MB]	Compression ratio
LZ4-r131		0.62	0.22	77	7.17
LZ4-r131	-9	8.86	0.22	54	10.21
LZ4 [1]	-9	10.06	0.23	54	10.21
Shrinker		0.69	0.27	73	7.56
LzTurbo [2]	-32	3.94	0.33	41	13.60
LzTurbo [3]	-32 -p1	5.35	0.54	41	13.6
lzop	-9	29.66	0.57	55	10.10
eXdupe		2.09	0.58	75	7.40
lzop		0.68	0.61	79	7.05
lrzip 0.621 [2]	-l	7.94	0.70	77	7.25
QuickLZ	-3	9.54	0.84	61	9.05
eXdupe	-x2	2.93	0.86	60	9.30
zlib-1.2.8 minigzip		9.90	0.91	48	11.63
NanoZip [-3]	-cf -M1670	1.00	0.94	67	8.35
compress [4]		4.17	0.98	56	9.95
GZIP	-9	12.46	1.07	48	11.70
GZIP	-5	5.29	1.09	49	11.42
lrzip 0.621 [5]	-l -p 1	8.26	1.11	77	7.25
Info-ZIP		9.74	1.23	48	11.64
GZIP	-1	2.50	1.25	58	9.65

Table 2.1: Results of the first text compression benchmark.

2.1.3 The Second Text Compression Benchmark

The second benchmark evaluates compression and decompression of the BNC's word, lemma, and ambtag attribute. The attributes are compressed and decompressed separately. The inputs are files containing one word of an attribute per line. All three attributes have the same number of lines, but their total sizes are different. The word, lempos, and ambtag attribute have 556 MB, 753 MB, and 443 MB respectively.

Manatee compresses the BNC's word attribute to 210 MB, lempos attribute to 203 MB, and ambtag attribute to 96 MB. Table B.4 in Appendix B shows sizes of the files for word, lempos and ambtag respectively. Considered are only files necessary to decode the whole attribute.

The results of the second benchmark are shown in Table B.1, B.2, and B.3 in Appendix B for word, lempos, and ambtag respectively. All results are sorted by the decompression time and the table cells are labeled in the same way as the results of the

first benchmark: grey background – worse compression ratio than Manatee. Only Table B.1 contains results of the Tornado 0.6 program. It was discovered during the evaluation of the second benchmark and the program was suppose to be much faster than the other programs. However, its results of the BNC's word attribute compression are below average which was the reason to not proceed with futher evaluations.

Manatee decompresses the BNC's word attribute in 10.3 s, lempos in 10.8 s, and ambtag in 8.4 s. It seems that many of the evaluated compression programs could replace the Manatee's encoding as they have better results. The following chapter explains why it is not sufficient to compare algorithms only by the decompression time of the whole text and what has to be compared.

2.2 Compression Algorithms and the Concordance Sort Operation

Manatee is rarely used to dump the whole corpus text. It can be even undesired in some cases, e.g. when we want to prevent from copying of the corpora, or when the corpora are extremely large. Therefore, the compression algorithms should be evaluated by performing operations that are commonly performed by the Manatee system. Such operations includes the frequency distribution and the concordance sort operation.

2.2.1 Segmentation

Basically, all operations executed by the Manatee system require random access to any word in the encoded text. The Manatee system uses Elias delta coding, which doesn't support random access. However, when a position of any encoded integer in a bit stream is known, then it is possible to decode all integers from the given location to the end of the stream. It would be memory consuming to store all positions of the encoded integers and because of that, the encoded integers are divided into segments of the same size, and only the positions of these segments are stored. Index of the segments provides relatively random access with a small overhead.

The algorithms in the benchmarks are not designed to provide random access to the words of the compressed text. The random access can be ensured by the segmentation, but the only way for most of the selected programs is to split an input text to segments, and compress these parts separately. As many of the programs uses already encoded data to encode the remaining uncompressed data, the segmentation can affect the size of the compressed data.

Table 2.2 shows how the size of segments affects the final size of the compressed data. The table presents sizes of the compressed BNC's word attribute with various segment size and different programs. It contains results of the GZIP [6], LZ-4 [7], and LzTurbo[8] compression.

All presented results have worse compression ratio than Manatee. The total size of the GZIP compression is increased by approximately 10 % in average by decreasing the segment size by half. The total size of the LZ-4 compression is increased by approximately 6 % by decreasing segment size by half. However, the LZ-4 has worse compres-

sion ratio than GZIP. The LzTurbo compression has the most significant degradation of the compression ratio. The compression with the segment of size 256 shrinks only 2 % from the input size. The futher evaluations were not performed as it was clear that it cannot be used.

The segment size always affects the size of the segment index file. Table 2.3 shows the sizes of segment index file with various segment sizes.

Segment size	Compressed with GZIP [MB] [6]	Compressed with LZ-4 [MB] [7]	Compressed with LzTurbo [MB] [8]
1024	253	397	326
512	273	424	410
256	299	453	547
128	337	485	
64	395	518	

Table 2.2: The BNC's word attribute divided into segments with different sizes and compressed with the GZIP, LZ-4, and LzTurbo compression.

Segment size	Segment index file size
1024	427.90 KB
512	855.80 KB
256	1.67 MB
128	3.34 MB
64	6.69 MB

Table 2.3: Segment size and size of the segment index file.

2.2.2 Benchmark of the Concordance Sort Operation

Benchmark of the concordance sort operation compares the performance of the concordance sort operation executed with different compressions and various segment sizes. The benchmark compares performance of the concordance sort operation with data compressed by the Manatee system, the GZIP[6] and LZ4[7] compression, and with uncompressed data (plain text). According to the second text compression benchmark, the LZ4 compression is the fastest decompression program, but it has worse compression ratio than Manatee. On the other hand, the best GZIP compression has better compression ratio of the word attribute than Manatee.

The concordance sort operation is performed with the concordance from the BNC corpus for the query **`[word="test"]`** and with criterion **`word/i 1>0~3>0`**. The query

6. with the option -9 (best compression)
7. source: <`https://github.com/bkaradzic/go-lz4`> (last commit: 74ddf825)
8. with the option -32

represents all occurrences of the word **`test`** and the criterion defines the right context as the following three words from each occurrence of the word **`test`**. The criterion also determines sorting of the context with case insensitivity.

Manatee performs the operation in 0.114 s with the default segment size (128 words in a segment). The compressed BNC's word attribute has 200 MB, lexicon has 7.2 MB, and index of lexicon has 3 MB. Table 2.4 shows how the performance of the operation is affected by the Manatee's compression with various segment sizes. The segment size affects the performance and size of the index of segments. The size of lexicon and size of the compressed text are not affected.

Table 2.5 and 2.6 present the results of the GZIP and LZ-4 compression respectively. The operation with the data compressed by GZIP has always worse performance than the operation with the data compressed by Manatee with same segment size. The operation with the data compressed by LZ-4 has achieved better performance than Manatee when the segment size is 64, but the size of the compressed file is almost equal to the size of the decompressed file.

Table 2.7 shows the performance of the concordance sort operation with uncompressed data. The results indicate theoretical performance limit of the sort operation, because the data decompression is for free. The operation with the uncompressed data has always better performance than Manatee with the same segment size. The speedup is approximately 1.426 in average.

Although, the results of the benchmarks compressing the whole text have better compression ratio and faster decompression, the results of the benchmark of the concordance sort operation show that it is not efficient to use segmentation with the selected algorithms to provide a random access to compressed data. This hypothesis is also supported by the results of the LzTurbo program, which has the best compression ratio among the programs in the second benchmark but worse compression ratio than Manatee when the text is divided into segments. LzTurbo is not open-source, so it is not suitable for the benchmark of the concordance sort operation.

Segment size	**Size of the compressed text [MB]**	**Time of the concordance sort operation [s]**
1024	200	0.407
512	200	0.231
256	200	0.147
128	200	0.114
64	200	0.088

Table 2.4: Performance of the concordance sort operation executed with data encoded by the Manatee system with various segment sizes.

Segment size	Size of the compressed text [MB]	Time of the concordance sort operation [s]
1024	253	1.027
512	273	0.699
256	299	0.533
128	337	0.430
64	396	0.395

Table 2.5: Performance of the concordance sort operation executed with the data compressed by GZIP with the option -9 (best compression).

Segment size	Size of the compressed text [MB]	Time of the concordance sort operation [s]
1024	397	0.566
512	424	0.298
256	453	0.179
128	485	0.115
64	518	0.084

Table 2.6: Performance of the concordance sort operation executed with the data compressed by LZ-4[7].

Segment size	Size of the compressed text [MB]	Time of the concordance sort operation [s]
1024	556	0.262
512	556	0.157
256	556	0.106
128	556	0.079
64	556	0.069

Table 2.7: Performance of the concordance sort operation executed with the uncompressed data (plain text).

2.3 Compression of Integers with a Lexicon Mapping Integers to Strings

According to the results of the benchmark of the concordance sort operation, the selected compression programs are not suitable as a replacement of the current Manatee's compression algorithm. However, none of the evaluated algorithms use the same approach as the Manatee system – compression of integers with a lexicon mapping integers to strings.

D. Lemire and L. Boytsov [12] claim "variable byte is twice as fast as Elias gamma

and delta coding", but their primary goal is to "decode billions of integers per second". They discuss and compare several encodings that should be faster and could have better compression ratio than Elias delta coding.

There is a possibility that the current Manatee's text compression could be improved just by using different encoding of the integers. This thesis do not aim to find a faster compression than the current compression algorithm, but to get rid of the lexicons mapping integers to strings. Nevertheless, a short comparison of encoding of integers is provided to lay a groundwork for a future research.

The short comparison evaluates two algorithms: BP32 and FastPFOR. According to the results of D. Lemire and L. Boytsov [12], both algorithms have one of the best compression ratio, but rather average decoding speed.

The BP32 scheme stores sequences of 32 unsigned 32-bit integers. The algorithm finds the minimum bit width needed to store each integer in the sequence, and then all integers of the sequence are stored to a block, in which each integer uses the computed bit width. The block's bit width is stored in a meta block which contains bit widths of the following 4 blocks of integers. [12]

The FastPFOR scheme works in a similar way to BP32. It also computes the minimum bit width needed to store each integer in a sequence of integers, but it tries to enhance compression by using a list of exceptions. The exceptions are used when it is advantageous to store only the lowest bits of integers in a sequence to a block and the remaining bits store as the exceptions. [12]

The performance of the concordance sort operation using data compressed by BP32 and FastPFOR are showed in Table 2.8 and 2.9, respectively. The BP32 scheme achieves better performance than Elias delta coding with the same segment sizes. The performance is better even with 4 times bigger segment size compared to the performance of the operation with data compressed by the Elias delta coding and the default segment size (128 words in a segment). However, the compression ratio is slightly worse – approximately by 4 %. The FastPFOR scheme also achieves better performance than Elias delta coding with the same segment sizes, and the compression ratio is also slightly worse – approximately by 3.4 %, so it is better the compression ratio of BP32.

Segment size	**Text file size [MB]**	**Time of the concordance sort operation [s]**
1024	231.89	0.166
512	232.31	0.115
256	233.15	0.082
128	234.82	0.069

Table 2.8: Performance of the concordance sort operation with the data compressed by BP32 and various segment sizes.

Segment size	Text file size [MB]	Time of the concordance sort operation [s]
1024	213.90	0.177
512	217.96	0.117
256	223.88	0.089
128	233.14	0.073

Table 2.9: Performance of the concordance sort operation with the data compressed by FastPFOR and various segment sizes. FastPFOR uses the default page size (65,536).

Chapter 3

Parallelization of the Selected Time-Consuming Operations

The frequency distribution and the concordance sort operation are time-consuming operations as they have to process the whole concordance before any results are produced. The performance can be enhanced by a parallelization of the operations. The parallelization can be performed locally on a single computer and globally on a cluster of computers.

3.1 Virtual and Split Concordance

The selected time-consuming operations are computed with a concordance. They can be parallelized by parallelization of the computations with the concordance. Two approaches are proposed: virtual and split concordance. The original implementation of the concordance is referred to as normal concordance.

The virtual concordance is based on an idea of dividing a large corpus into multiple smaller parts. An operation can be computed on each part separately, and the partial results can be merged together. Moreover, the parts of the corpus can be stored on different computers, so the virtual concordance enables parallelization on a single computer and a cluster of computers.

The split concordance uses a single corpus just like the normal concordance. They differ in the design of operations. The split concordance divides data of the concordance into splits and computes an operation concurrently with each part. Then, the partial results are merged. The concurrency enables parallism when a computer has more available processors.

3.1.1 Benchmarks of the Virtual and Split Concordance

The virtual and split concordance benchmarks evaluate the performance of the frequency distribution and the concordance sort operation with the virtual and split concordance. Both operations are executed with several queries from the BNC corpus that differ in the size of the results. Table 3.1 shows queries used in the benchmarks and the sizes of results of both operations. The concordance sort operation was performed with the criterion **`word/i 1>0~3>0`**, and the frequency distribution was performed with the criterion **`word 1>0`**.

Both operations were evaluated on one core and also on the maximum number of available cores (8 cores). The evaluation uses a single computer with the following hardware specification: 8 GB RAM, 256 GB SSD disk, and 2.0 GHz Intel Core i7 (I7-4750HQ) – 4 cores (plus 4 virtual cores) processor.

Query	**Size of the concordance = number of results of the concordance sort operation**	**Number of results of the frequency distribution**
`[word="Gauss"]`	60	15
`[word="recurrence"]`	500	68
`[word="enjoyment"]`	1,000	104
`[word="test"]`	11,040	1,265
`[word="said"]`	194,767	7,364
`[word="a"]`	2,040,346	76,277

Table 3.1: Queries used in the virtual concordance and split concordance benchmarks and number of results of the frequency distribution and the concordance sort operation.

Benchmark of the Virtual Concordance

The virtual concordance benchmark compares the performance of the virtual concordance on the BNC corpus divided into various number of parts. All parts are located on a single computer.

The results of the benchmark are in Appendix C: Table C.1 and Table C.2 for the concordance sort operation and frequency distribution, respectively. The performance is enhanced each time the number of parts is increased. The enhancement is achieved until the number of parts is less than 8. The evaluations with 8 parts have worse performance when the queries with rather small size of results are evaluated, but the performance is better when the queries with rather larger size of results and evaluated on all available cores. The evaluation with 9 parts have very similar performance as with 8 parts.

Benchmark of the Split Concordance

The split concordance benchmark compares the performance of the split concordance on the whole BNC corpus and various number of the splits.

The results of the benchmark are in Appendix D: Table D.1 and Table D.2 for the concordance sort operation and frequency distribution, respectively. The performance is enhanced each time the number of parts is increased. The enhancement is achieved until the number of parts is less than 8. The evaluations with 8 parts have mostly worse performance when the queries are evaluated only on one core, but better performance when the queries are evaluated on all available cores. The evaluation with 9 parts have very similar performance as with 8 parts.

3.1.2 Comparison with the Normal Concordance

The concordance sort operation and frequency distribution are evaluated with the normal concordance and the same queries as used in the virtual and split concordance benchmarks. The evaluation uses also the same environment. Table 3.2 and 3.3 show evaluation of the normal concordance and the results are compared with the results of the virtual concordance with 8 parts and the split concordance with 8 splits.

The benchmark of the virtual and the benchmark of the split concordance indicate that the performance with 8 parts and 8 splits reaches almost the maximum performance. Although, more than 8 parts and splits can slightly enhance the performance, the performance of some queries can be degraded.

Both, virtual and split concordance, have better performance than the normal concordance when the operations are executed with all available cores. In some cases, the virtual and split concordance have also better performance when running on one core.

The virtual concordance has better performance in more cases than the split concordance, and also the benchmarks show that the virtual concordance have better performance with lower number of parts when compared to the number of splits.

Query	Number of available processors	Performance of the normal concordance	Performance of the virtual concordance (8 parts)	Performance of the split concordance (8 splits)
`[word="Gauss"]`	1	0.014	0.015	0.018
	8	0.013	0.006	0.004
`[word="recurrence"]`	1	0.114	0.099	0.119
	8	0.118	0.034	0.030
`[word="enjoyment"]`	1	0.275	0.253	0.281
	8	0.270	0.047	0.057
`[word="test"]`	1	1.368	1.408	1.429
	8	1.279	0.233	0.260
`[word="said"]`	1	6.439	6.394	6.339
	8	5.625	1.206	1.300
`[word="a"]`	1	27.909	28.435	28.727
	8	24.753	7.067	7.606

Table 3.2: Performance of the concordance sort operation compared between the normal, virtual (8 parts), and split (8 splits) concordance.

3.2 MapReduce

MapReduce is a programming model inspired by the functional programming which makes it easy to parallelize execution on a large cluster of commodity computers. The model provides an abstraction of computations with large amount of data in a dis-

Query	Number of available processors	Performance of the normal concordance	Performance of the virtual concordance (8 parts)	Performance of the split concordance (8 splits)
`[word="Gauss"]`	1	0.009	0.010	0.012
	8	0.010	0.003	0.003
`[word="recurrence"]`	1	0.101	0.077	0.099
	8	0.118	0.034	0.030
`[word="enjoyment"]`	1	0.252	0.215	0.264
	8	0.250	0.038	0.051
`[word="test"]`	1	1.212	1.213	1.184
	8	1.209	0.206	0.237
`[word="said"]`	1	4.852	4.941	4.972
	8	4.536	0.796	0.985
`[word="a"]`	1	10.021	10.965	9.824
	8	9.184	2.290	2.910

Table 3.3: Performance of the frequency distribution compared between the normal, virtual (8 parts), and split (8 splits) concordance.

tributed environment. The aim is to hide details of parallelization and problems related to using a cluster of computers – such as data distribution, fault tolerance, or load balancing. [28]

The model enables to define a computation as a map and reduce functions, which are present in many functional languages (e.g. Lisp, Haskell). The map function is applied to records of an input (e.g. lines, words), and emits intermediate key-value pairs. The intermediate pairs with the same keys are grouped and processed by the reduce operation. This approach makes it easy to express distributed algorithms as simple computations, and solve fault tolerance with simple re-execution of the operations. [4]

3.2.1 MapReduce and Manatee

As Tom White [28] claims "MapReduce is a complement to the traditional rational database management system (RDBMS) in many ways". RDBMS has a lot in common with the Manatee, e.g. strict data schema, indexed data, and interactive queries, but probably the only thing that have Manatee and the MapReduce run-time in common is how they treat the data: write once and read many times [28]. Although, it appears that Manatee is not suitable for MapReduce frameworks, the operations match the MapReduce programming model.

The concordance sort operation and frequency distribution can be implemented as MapReduce tasks with the virtual or split concordance. Both operations are evaluated with a concordance and criteria. Parallelization in a cluster of computers requires to divide a corpus into smaller corpora that are stored in the cluster.

The map function has to receive a part of a divided corpus, query, and criteria. If an

implementation uses the virtual concordance, the map function can receive more than just one part of the divided corpus. The function creates the concordance, executes the operation, and emits the results. If a split concordance is used, the map function must divide the concordance into splits. The implementation can contain two phases of the map phase. The first phase creates and divides the concordances. The second phase can receive the splits, execute the operation, and emit the results.

MapReduce Model of the Frequency Distribution

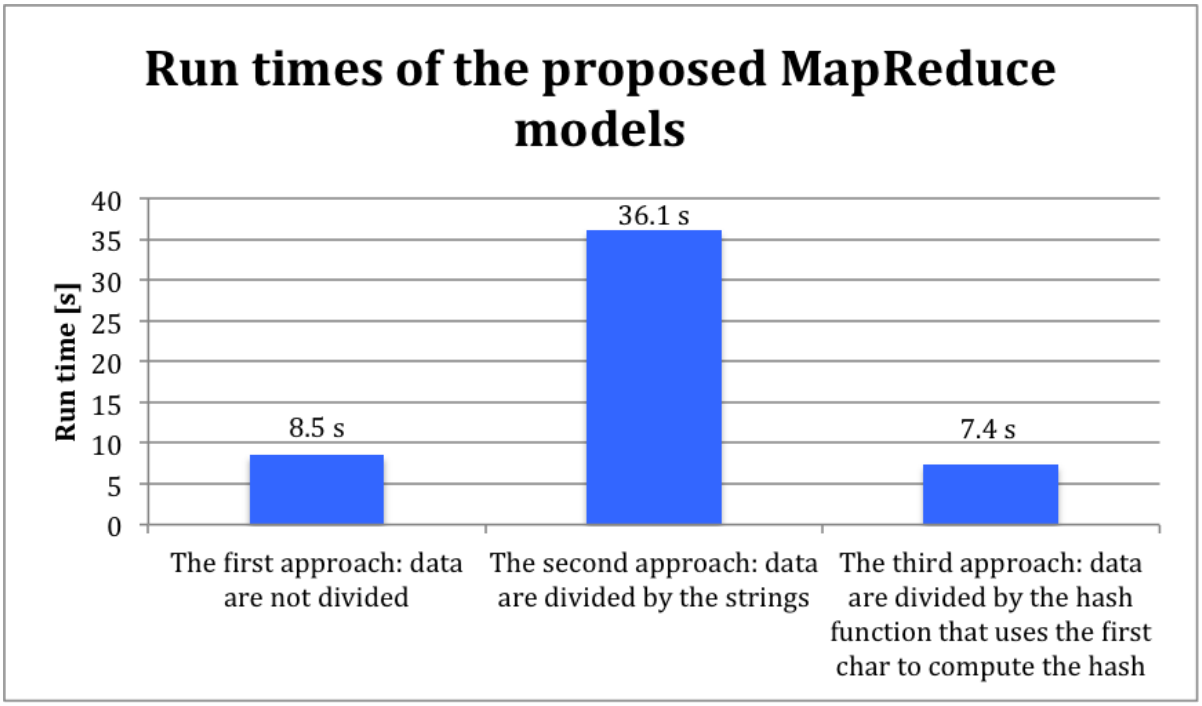


Figure 3.1: Run times of the proposed MapReduce models of the frequency distribution computed from a concordance with large amount of data and executed with the Glow framework.

The frequnecy distribution is computed from a concordance and criteria. The criteria are evaluated for each item of the concordance and the result of the frequency distribution is a list of pairs of the evaluated criteria and number of its occurrences.

The MapReduce model of the frequency distribution can be quite simple. The map phase can create concordances and execute the operation. The results are emitted. The reduce phase receives lists of pairs. They can be merged together and the merged results can be sorted by the frequencies. This model uses only one reducer, which can be a bottleneck of scalability.

The proposed first approach can be enhanced by using more reducers. The map function can be changed to iterate the created list and emit the pairs. The emitted pairs

can be partitioned by the key, so all pairs with the same key will be given to the same reducer. The maximum number of reducers is equal to the number of different keys. The reduce phase has two stages. The first stage of the reduce phase sums the frequencies for each group of pairs with the same key, and emit the results. The second stage merges the results together and then sorts them.

The second approach emits huge number of pairs – at most **`NUMBER_OF_MAPS * NUMBER_OF_UNIQUE_STRINGS`** in the map phase, and **`NUMBER_OF_UNIQUE_STRINGS`** in the reduce phase. It can be optimized by using a hash function, that creates a hash code from the string. The map function emits all pairs with the same hash as one pair. The emitted pair has the hash code as the key, and all appropriate pairs as the value. The advantage of using the hash function in the map phase is that the reducers will receive smaller number of pairs. The reducers can merge the lists with the same keys and sort them by the frequencies. Then, the sorted lists are merged together in such way that the output list will be sorted by the frequencies.

All three approaches were evaluated with the Glow framework [15]. The third approach uses a simple hash function that returns the first character of a given string as the hash code. The Glow framework is an open-source MapReduce system written in Go [5]. The framework is more described in the section 3.2.3. Figure 3.1 shows the measured performance of the specified approaches. They were evaluated on huge amount of data in a cluster of 20 computers. The used concordance had 87,159,527 items and the frequency distribution produced 1,056,289 results. The comparison suggests to use a hash function to divide large amount of data when the frequency distribution is computed using the Glow framework.

MapReduce Model of the Concordance Sort Operation

The concordance sort operation is more complex than the frequency distribution. The operation uses a concordance, criteria, and query. The criteria are evaluated for each item of concordance, and the concordance is sorted by the evaluated criteria. The result of the concordance sort operation is a sorted concordance. The sorted concordance is presented to a user just like any other concordance – in form of KWIC lines.

The Bonito web interface lays out the KWIC lines to separate pages and displays one page at a time. The KWIC lines are based on the KWIC format, which is customizable and can be changed. The web interface enables to display the following or previous pages, jump to any page, and provides a sort index that can be used to jump to a page containing the first lines with a specified key. The sort index is usually built from the first characters of the evaluated criteria.

The sort can be implemented as two-phase merge sort. The map phase passes the evaluated criteria to the reduce phase, which merges them. Each evaluated criteria has to contain additional information that can be used to identify an original concordance. The created concordances must be stored as they are used to compute the requested KWIC lines. The identificator is used by a client that gets the merged results and asks for the KWIC lines of items that are present at a requested page. The MapReduce frame-

work can be used to store the concordances and also to compute the KWIC lines.

The described model is simple, but the amount of merged data can be significantly reduced. It is not always necessary to merge all results of the map functions, e.g. when the first page is required, only the first items could be merged, then when the second page is required, only the next items could be merged. The results of the map functions can be divided into segments of the same size and stored. The merge can be done on demand only for those segments that are necessary to merge to get the results for a requested page.

The enhancement solves the common case – displaying the first page and iterating to the next and previous pages. Jump to the middle page can be optimized by building a sort index for each concordance. However, the optimized solution is beyond a simple MapReduce model and because of that, it won't be implemented.

3.2.2 Hadoop

Hadoop is an open-source framework implementing MapReduce written in Java. The idea is derived from Google's MapReduce and Google File System (GFS) [7]. Apache Hadoop was created by Douch Cutting, the author of Apache Lucene – text search library. Originally, it was part of Apache Nutch, an open source web search engine. The framework is used by more than 150 companies such as Yahoo!, Facebook, Twitter, IBM, LinkedIn, The New York Times, or Adobe [21].

The framework is able to run applications on large cluster of computers, e.g. in January 2007 – cluster with 900 nodes, in March 2009 – 17 clusters with a total of 24,000 nodes. A cluster can consist of cheap commodity hardware – commonly available hardware. It can process large-scale data quite fast, e.g. in April 2008 Hadoop broke a world record as the fastest system sorting a terabyte of data.

Data are stored to and loaded from HDFS (Hadoop Distributed File System). HDFS is designed to store large files (hundreds of MB, GB, or TB in size) with streaming data access pattern. It is optimized for a high throughput of data which can be in cost of higher latency. Data are stored to HDFS blocks which are 64 MB by default. Block is the minimum amount of data that can be read or write.

Hadoop is designed for batch processing of the whole files, but Manatee is based on the interactive queries and random access to the files. The framework is not suitable for tasks provided by Manatee. On the other hand, there are many Hadoop related projects, called the Hadoop ecosystem, and some of them (e.g. HBase) could probably meet Manatee's requirements for the random data access and suit for the distributed computing of Manatee's operations. However, such research is outside of scope of this thesis as it would probably require enormous changes in Manatee's persistent layer and there is no garantee that such system would have better performance than the distributed system presented in this thesis.

More information about Hadoop can be found in [28].

3.2.3 Glow

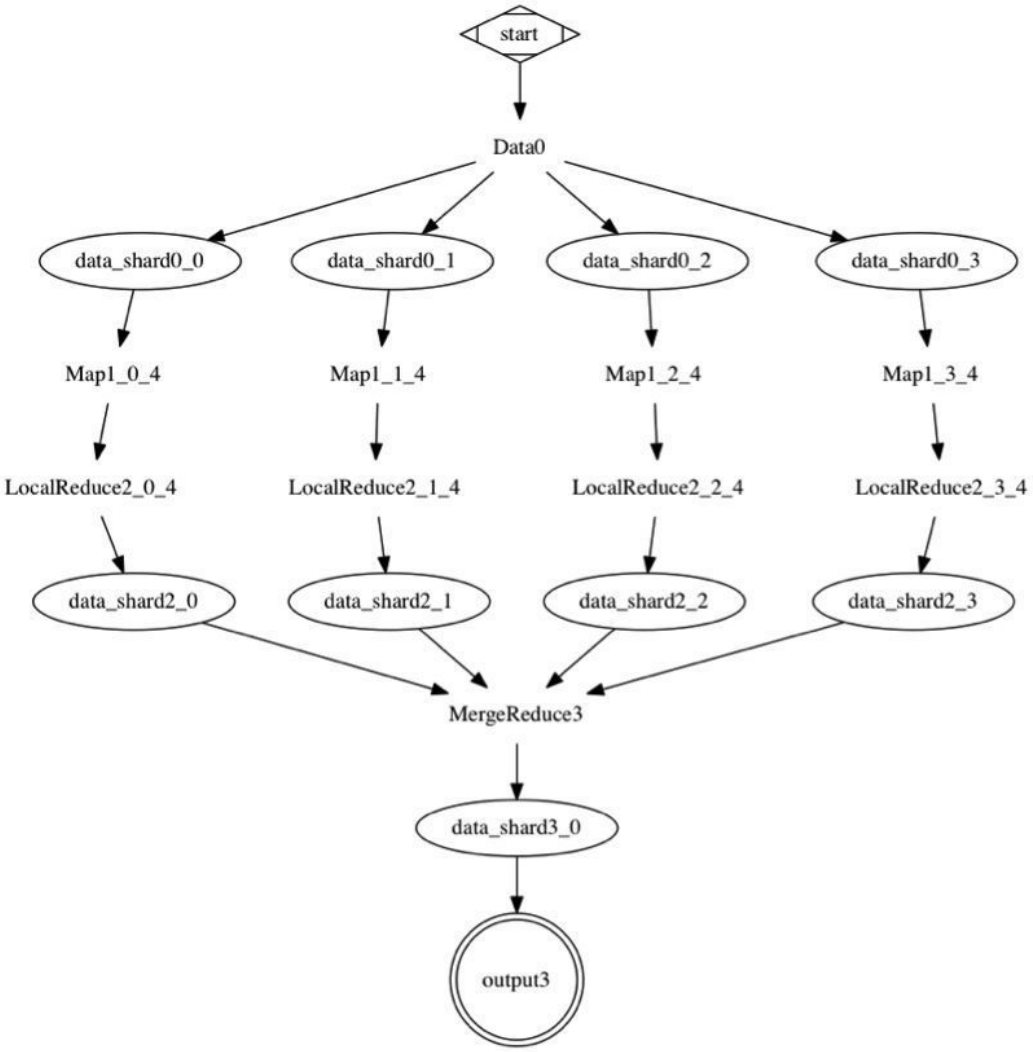


Figure 3.2: Example of a directed acyclic graph created by a driver application.

Glow is an open-source MapReduce system implemented in Go (also called Golang, [5]) similar to Hadoop [28]. The primary goal of the system is to provide simple and scalable framework supporting the MapReduce programming model. Glow has been developed since 2015 and so far it does not handle fault tolerance, monitoring of computers in the cluster, or error handling. An applications using Glow can run in 2 modes: standalone or distributed.

The distributed mode uses the master/slave architecture. The master must run only on one computer, but slaves, called agents, can run on one or more computers. The master provides resource management. The agents report resource usage to the server by heartbeats.

An application using the Glow framework running in the distributed mode can act either as a driver or an executor. The mode of the application is specified via command line arguments. The driver handles the distributed computing. It divides tasks into directed acyclic graph (see example at Figure 3.2). The graph is optimized to minimize streaming of data to a disk, and is used to create tasks groups that consist of subsequent tasks. The driver asks the Glow master server for the resources required by each tasks group and the server checks whether there are available such agents that fullfil the specified resource requirements. If so, the master assigns the agents to the driver.

The driver sends groups of tasks to the assigned agents. Each contacted agent receives tasks, locations of the input data, and binary copy of the driver, which will run in the executor mode. The executor reads the data from network channels, executes the tasks, and writes the result to local agent.

More information about the Glow framework can be found at [15].

Glow and Implementation of the Frequency Distribution

The section 3.2.1 concludes that the concordance sort operation and retrieval of the sort concordance page is beyond a simple MapReduce model. Therefore, only the frequency distribution is implemented with the MapReduce framework. The implemented model of the frequency distribution is equivalent to the best model described in the section 3.2.1. The implementation uses the virtual concordance because it has better results on a single computer than the split concordance.

Glow in Standalone Mode Benchmark

The implementation of the frequency distribution using the Glow framework is evaluated in the same way as the virtual concordance. The benchmark uses the same configuration, parameters, and environment. The executed program runs in the standalone mode.

Table E.1 in Appendix E shows results of the benchmark. They are very similar to the results of virtual concordance. The virtual concordance has better performance, but the difference in the performance shrinks as the queries produce large number of results.

Glow and Data in a Cluster

The proposed design of the distributed computation on a cluster of computers uses a large corpus divided into smaller corpora. The smaller corpora are stored on computers of the cluster.

The Glow framework does not support storing data to or locating data in a cluster. Data are either available from all nodes or they are transmitted from a driver (client application). The framework supports definition of a node's location and its capabilities. The computer's location is naturally defined by its address and port number, and Glow also enables grouping of computers into data centers and racks. The node's capabilities is a list of memory size, number of CPU cores, CPU level (relative computing power of a

single CPU core), and the upper limit of the number of executors running concurrently on the agent [16]. A driver can define name of the preferred data center and rack, but allocation of such agents is not guaranteed.

It is required to support propagation of information about corpus parts located on each computer of the Glow cluster and to allow a driver to specify demanded corpus parts. Glow is an open-source software, so it can be freely modified and extended to match our design.

The required corpus parts can be stored as an attribute of a tasks group, because the required resources are computed from the tasks groups. The attribute will be set only for the tasks groups containing a map task, because the corpus parts are used only in the map phase.

The agents have to report the available corpus parts to the master server. The available corpus parts can be specified via a command line flag or an environment variable. The driver can specify what corpus parts are needed to execute tasks groups and the master assigns only such agents that have such resources on their computers.

Chapter 4

Implementation

The implemented system uses large corpora divided into smaller parts that are located on computers of a cluster. On each computer runs a server that has an access to compiled corpus parts. The implementation has the client/server architecture, but unlike the traditional communication model of the architecture, the model of the implemented system involves one client and multiple servers. A client contacts the servers that have a required data and asks them to perform some operations. The results are returned to the client, which processes the results and displays the output to a user.

The system is written in Go and extends the latest version of the Manatee system implemented in Go. The Manatee system is also written in C++, but the Go version provides faster processing of corpus queries and it is intended to use only the Go version in the future. [23]

4.1 Server

Server is a REST service [24] communicating over the HTTP protocol with clients. It exposes an interface that provides an access to the corpus parts located on the computer, on which the server application is running. The interface offers the following operations:

- get information about corpora – returns total size of the specified corpora.
- create a virtual concordance – accepts a request containing names of corpus parts, query, and concordance name. The server computes the virtual concordance for the queries from each corpus part. The created concordance is stored with the specified name on a server. The response is the size of the created concordance.
- get a concordance page – a request contains a concordance name, page number, page size, and KWIC format. The server loads the requested concordance from a local storage and computes the requested page. The page has form of KWIC lines with the specified KWIC format. The result is a list of the computed KWIC lines.
- compute the frequency distribution – the frequency distribution is computed on a concordance specified by its name and with criteria defined in a client's request. The server returns results of the frequency distribution. The results are lexicographically sorted by the evaluated criteria.

- sort a concordance – a client defines a name of concordance to be sorted and criteria of the concordance sort operation. The appropriate concordance is loaded and sorted by the results of the criteria evaluation, referred to as sort data. The sorted concordance and the sort data are divided into segments of the same size and stored. Each segment contains the same amount of items of the sorted concordance as the amount of items of the related sort data. An index of segments is built to enable a random access to the stored segments. The server's response to the request is the number of segments and a sort index. The sort index is created from the sort data and is usually built from the first characters of the sort data.

- get a segment of a sorted concordance – a client sends a request with a segment number, concordance name, and KWIC format. The server loads the specified segment of the sorted concordance, computes the KWIC lines with the specified format, and returns the computed KWIC lines and the sort data of the requested segment.

4.2 Client

Client handles the distributed computation of a concordance, the frequency distribution, retrieving of a specified concordance page, and the concordance sort operation and retrieving of a specified page of the sorted concordance. All operations can be executed only when the client knows locations of the parts of a divided corpus. They are defined in a configuration file.

The configuration file defines locations of parts of a corpus. The file consists of blocks that start with a corpus name followed by list of servers with names of the corpus parts located on each server. The servers are described by their URLs and port numbers. The corpus parts on the server should have names with the same prefix and subsequent suffix numbers, so the names can be expressed by the common prefix and range of the suffix numbers. For example, the corpora names **`bnc-part1`**, **`bnc-part2`**, and **`bnc-part3`** can be described with the prefix **`bnc-part`** and range from **`1`** to **`3`**.

It is expected that the suffix numbers denote the order of parts, and the list of servers is sorted by the order of parts. It means that the first part of a divided corpus is located on the first listed server and other parts located on the same server have greater suffix numbers.

4.2.1 Workflow of Receiving a Concordance Page

The client sends a request to create a virtual concordance to all servers containing any part of a specified corpus. The servers return sizes of their concordances. The client determines which concordance or concordances contain results of the requested page, and asks for the relevant pages. The pages are joined together to one page and presented to a user.

4.2.2 Workflow of the Frequency Distribution

The distributed computation of the frequency distribution has relatively easy workflow. First, all servers containing any part of a specified corpus are requested to create a concordance. Then, the servers are asked to perform the frequency distribution. They produce lists of pairs of the evaluated criteria as a key and number of the key's occurrences (frequency) as a value. The keys of the lists are lexicographically sorted. The client merges the entries of the lists with the same key and sums the frequencies. The merged results are sorted by the frequencies and displayed to a user.

4.2.3 Workflow of the Concordance Sort Operation and Retrieving of a Specified Page

The concordance sort operation and retrieval of a specified page is the most complex operation that is present in the implemented system. The workflow begins with the same steps as the workflow of the frequency distribution: contact all servers that have any part of a specified corpus, send requests to create the virtual concordances and execute the operation (sort a concordance).

Results of the sort a concordance are sort data, sorted concordance, and sort index. The client merges the sort data and computes which items of the sort data correspond to the specified page requested by a user. The items are used to determine list of servers that have the concordances involved in computation of the specified page. These servers are asked to compute KWIC lines that will be used to construct the requested page. The computed KWIC lines are returned together with the related sort data. The returned sort data are merged in such way that the results are lexicographically sorted. The returned KWIC lines are merged in the order denoted by the merged sort data and the constructed page is presented to a user.

Merge of the sort data can be optimized to significantly reduce amount of the merged data. The optimization is briefly described in the section 3.2.1. It is easy to optimize retrieval of the first page, because the merge phase can merge all sort data from the beginnings and proceed with the merge only until the the number of merged items is lower than the size of the first page. Retrieval of the second page is similar, but it merges two times more data than the retrieval of the first page, or just the same amount of data when the second page is called as a retrieval of the next page of the first page. It is easy to compute the next or previous pages of any page when the client had already computed the page that precedes or follows the requested page. The problem is jumping to a random page.

The jumping to a random page is similar to loading a page by merging the sort data from the beginnings. It is optimized by using the sort index, which is computed from the sort data. The sort index enables to reduce amount of the merged data when the random page is requested.

The sort indices from the servers can be merged together to get a global sort index. The global sort index can be used to find a key that has its first occurence on the re-

quested page, or if such key does not exists, a key that is just present on the page. The sort indices of the servers are searched for the selected key which can result in one of the following cases: 1. an index contains the key; 2. an index does not contain the key but contains greater keys; 3. all keys of the sort index are lower than the selected key. If an index contains the key, then the first segment containing the key is loaded. If the sort index does not contain the key but contains greater keys, the first segment with the first greater key, referred to as greater segment, is loaded. If all keys of the sort index are lower than the selected key, then the last segment of a sorted concordance, also referred to as lower segment, is loaded.

Iterators

Data of segments and merging of the data are managed by iterators. The loaded segments are iterated by segment iterators. The items of loaded segments are merged by line iterators that use the segment iterators to access the segments. The line iterators are used by a page iterator to construct a requested page.

Segment Iterator

The items of segments are accessed via segment iterators. The iterators are initialized with the loaded segment. The current value of an segment iterator is set to the first item with the selected key, or if a greater segment was given, it is set to the first item with the selected greater key, or if a lower segment was given, the current value is set to the last item. The current value of the iterator is a pair of the KWIC line and sort data that correspond to the current item of the segment.

Segment iterator provides methods to load the next and previous items of the segment. The iterator handles loading of the next and previous segments when it is necessary.

Line Iterators

The segment iterators are used by line forward iterators and line backward iterators. Line iterators group together the segment iterators and merges the sort results to calculate the next or previous line.

The line forward iterator groups the segment iterators with the segments containing the selected key and with the greater segments. The selected segment iterators are inserted into a min-heap that arranges the segment iterators by their current values of the sort data. The line forward iterator uses the top item of the min-heap as its current value. The iterator provides next line method. It calls the method of the top segment iterator of the min-heap to load the next item, and then fixes the heap.

The line backward iterator is similar. It groups the segments iterators with the segments containing the selected key and with the lower segments, and puts them into a max-heap. The line iterator uses the top item of the max-heap as its current value.

The iterator provides previous line method, which calls the method of the top segment iterator of the max-heap to load the previous item, and then fixes the heap.

Page Iterator

Page iterator is an iterator of pages of a sorted concordance. The initialization initalizes the page iterator to a specified page. It uses the global sort index to find the key of the specified page as it was described. The position of the selected key in the first page containing the key is computed and the line iterators are initialzied to the computed position. First, the page iterator uses the line backward iterator to get lines from the computed position to the beginning of the first page with the selected key. Then, the page iterator uses the line forward iterator to get lines from the computed position to the end of the first page with the selected key. After the first page with the selected key is computed, the page iterator uses the line forward iterator to load the next pages until the requested page is loaded.

The iterator provides methods to load the next page, the previous page, and to get the current page. The current page is list of the KWIC lines that correspond to the page of the sorted concordance.

4.3 Format of the Transmitted Data

Format of the transmitted data affects the size of the transmitted data, and therefore, the time of the transmission is affected. The performance of data transmission is also affected by the time to encode and decode the data.

Requests to the server uses the JSON[1] format, which is a human-readable and language independent data format derived from JavaScript [3]. JSON is also used as format of the responses that do not contain large amount of data, e.g. create a concordance returns only the size of the created concordance.

It it expected that the frequency distribution can produce huge amounts of data, so the encoding of the data can significantly affect the performance of the data transmission. Besides JSON, there are multiple other formats that can be used to transmit data:

- Protocol Buffer is Google's language-independent data serialization. The serialized information must be first defined as a protocol buffer message in a separate file. The messages are compiled with the compiler provided by the Protocol Buffer. The compiler generates necessary code written in a selected langauge. [11]
- Gogoproto is an extension of the protocol buffer for Go. It provides additional features targeted to the Go language such as fast marshalling and unmarshalling, more cononical Go structures, or generation of tests and benchmarks. Fast marshalling and unmarshalling avoid using of reflection by generating explicit methods for the serialization and deserialization. [22]

1. JavaScript Object Notation

- Gob is a Golang specific data serialization. It does not require definition of an interface or separate compilation of a message. The serialization creates streams that are self-describing. Its design is heavily influenced by Google's Protocol Buffer, but some features are deliberately missing to keep the implementation simple. Gob has expensive compilation of a data type, so it is recommended to use a single Encoder multiple times. [19]

4.3.1 Benchmark of the Data Transmission with Different Formats

The benchmark compares the described formats used to transmit data: JSON, Gob, Protocol Buffer, and Gogoproto. Gob is evaluated in 2 variants, denoted as Gob* and Gob**, to measure how expensive is a compilation of a data type. Gob* creates a new instance of Encoder and Decoder each time a data type is encoded and decoded – each encoding requires compilation of the data type. Gob** uses one shared instance of Encoder and Decoder. Serialization with Gogoproto uses fast marshalling and unmarshalling, casting types to match the Manatee's type system, and non-nullable feature specifying that a field of the Protocol Buffer messages must be always set – cannot have the null value.

The benchmark measures performance of the serialization and deserialization, and size of the encoded data. The input dataset consists of the results of the frequency distribution and the concordance sort operation (the sort data and KWIC lines). The KWIC lines have the KWIC format that shows at most 40 characters as the left context and at most 40 characters as the right context, the KWIC, and the beginnings and ends of sentences and paragraphs. The results of the concordance sort operation are divided into chunks of 50 elements because the server also divides the results into chunks.

The frequency distribution and the concordance sort operation are computed with the criteria **`word 1>0`** and **`word/i 1>0~3>0`**, respectively. They are computed from various concordances from the BNC corpus. Encoding of such results are, almost in all cases, too fast (below 1 second), so the benchmark measures repeated encoding of the whole input.

The measured results are presented in Appendix F. According to the results, the best format is Gogoproto. Gob* has always worse performance than Gob**. The difference between them is more significant when the size of data is smaller. Gob** achieves the smallest size of the encoded data (when the data type is already compiled). JSON has the worst results almost in all cases – except the case when small amout of data are serialized (encoded as hundreds of bytes). Moreover, the JSON format has approximately 2–3 times bigger size of the encoded data.

Gogoproto serializes the data with the same compression ratio as Protocol Buffer, but Gogoproto has better performance. It is caused by the features of the fast marshalling and unmarshalling, and by using additional features such as type casting and non-nullable fields, which makes a structure representing a protocol buffer message identical to a structure representing an application's data. It is much easier and faster to convert one structure to another, when the structures are identical.

Table 4.1 shows the results of the benchmark as average ratios to Gogoproto. The

sizes of the encoded data by Gob** were measured after the data types were already compiled. Because of that, they are denoted with ***. The difference between streams with and without the compiled data type is 86 B for the data type of results of the frequency distribution, and 309 B for the data type of results of the concordance sort operation.

Based on the results of this benchmark, the implementation uses Gogoproto to serialize and deserialize results of the frequency distribution, segments of the results of sort operation, and a concordance page.

Format	**Transmission of the frequency distribution data**		**Transmission of the segment containg the sort data and LWIC lines**			
	Average ratio of the time	**Average ratio of the size of encoded data**	**Average ratio of the time**	**Average ratio of the average size of encoded data**	**Average ratio of the minimum size of encoded data**	**Average ratio of the maximum size of encoded data**
JSON	9.126	3.119	4.383	1.873	1.878	1.934
Gob*	6.676	1.062	2.786	1.023	1.063	1.002
Gob**	3.046	***0.925	1.948	***0.982	***0.983	***0.975
Protocol Buffer	2.465	1	1.483	1	1	1

Table 4.1: Results of the benchmark of the data transmission with different formats. The results are expressed as average ratios to Gogoproto.

4.4 Future Development

The implemented system provides methods to create a virtual concordances, get a concordance page, sort a concordance, get a page of a sorted concordance, and compute the frequency distribution. Some features are missing such as selection of collocation candidates or computation of histogram data. Implementation of the missing features should be flawless, because the absent functionality is simplier or not more complex than the implemented concordance sort operation and retrieval of a page of a sorted concordance.

Chapter 5

Evaluation

The evaluation of the implemented system is executed on the Nymfe cluster, the Aurora server, and the Arachne server. The Nymfe cluster consists of the Nymfe computers – computers in the Computer lab of Faculty of Informatics at Masaryk University. The lab has more than 100 computers and they are available to everybody with the access to the faculty network or to the Computer lab. It is not possible to get an exclusive access to the computers and the evaluation can be affected by additional processes running on the computers.

The implementation is evaluated on the enTenTen 2012 corpus. It has more than 11 billions of words and it is from the family of TenTen corpora – corpora with size of 10 billion words. They were created by web crawling and additional postprocessing, e.g. to remove duplications. [8]

The corpus is divided into 130 parts that are stored on the first 65 Nymfe computers: Nymfe01–Nymfe65. Each of the 65 Nymfe computers holds 2 subsequent parts. The corpus divided into 130 parts is also referred to as 130 parts corpus.

The Nymfe computers provide very similar environment. All have the GNU/Linux operating system, 4 cores processors, 16 GB of RAM, but they have different processors. Table 5.1 lists the computers and their processors of the Nymfe cluster.

Clients executing tasks on the Nymfe cluster were executed on the Aurora server. The server is connected to the network of Faculty of Informatics and it has 16 GB of RAM and 8 cores Intel® Xeon® CPU X5355, 2.66GHz.

Other programs and evaluations were executed on the Arachne server. The server is in the network of Masaryk University. The Arachne server uses RAID 6, and has 32 GB of RAM and 8 cores Intel® Xeon® CPU E5-2680 v3, 2.50GHz.

Computers	**Processor**
Nymfe01–Nymfe22	Intel® Core™ i5-4590 CPU, 3.30GHz
Nymfe23–Nymfe74	Intel® Core™ i5-3470 CPU, 3.20GHz
Nymfe75–Nymfe86	Intel® Core™ i5-3330 CPU, 3.00GHz
Nymfe87–Nymfe105	Intel® Core™ i5-4690 CPU, 3.50GHz

Table 5.1: Computers of the Nymfe cluster and their processors.

5.1 Comparison with the MapReduce Framework

The frequency distribution is described also as a MapReduce model in the section 3.2.1. The map phase uses Manatee to compute the frequency distribution and passes the results to the reduce phase. The reduce phase merges, and then sorts the results. The model can be implemented with the Glow framework modified to locate corpus parts in a cluster of computers. The modification is described in the section 3.2.3.

The performance of the frequency distribution implemented with the Glow framework is compared with the performance of the implemented system. The evaluation was executed in the Nymfe cluster. The Glow agents run on the Nymfe1–Nymfe65 computers and the Glow master server run on Nymfe66. The client was executed on the Aurora server. The frequency distribution was evaluted with the queries listed in Table 5.2 and criterion **`word 1>0`**.

Table 5.3 shows the performance of frequency distribution implemented with the Glow framework and in the implemented system. The implemented system has better performance when the queries has rather small number of results and the Glow framework is more powerful to process queries with quite extreme huge number of results. The bottleneck of the implemented system is merging and sorting of the results on a client's computer, but it is much simplier as executing the reduce phase on multiple computers.

The evaluation of the application using the Glow framework was quite complicated as Glow does not provide error handling or fault tolerance. The operation sometimes produced unexpected results and the cluster of the Glow agents seemed to be unstable.

Query	Number of results of the frequency distribution	Number of occurences in the corpus
`[word="Gauss"]`	497	2,132
`[word="recurrence"]`	1,580	28,927
`[word="enjoyment"]`	48,41	157,287
`[word="test"]`	33,100	1,625,427
`[word="said"]`	208,676	10,842,497
`[word="a"]`	1,700,427	241,926,311
`[word="the"]`	3,716,817	547,226,436

Table 5.2: Queries used in the evaluations of the implemented system and sizes of the results from the enTenTen 2012 corpus.

5.2 Comparison with the Original System

The comparision of the implemented system with the original system contains performance evaluation of the frequency distribution, the concordance sort operation and re-

Query	Time of the frequency distribution by Glow	Time of the frequency distribution by the implemented system
`[word="Gauss"]`	2.168	1.760
`[word="recurrence"]`	2.409	2.260
`[word="enjoyment"]`	2.492	2.325
`[word="test"]`	4.210	3.797
`[word="said"]`	5.194	5.363
`[word="a"]`	10.851	16.190
`[word="the"]`	15.594	29.790

Table 5.3: Performance of the frequency distribution by the Glow framework and the implemented system.

trieval of the first page of a sorted concordance, and computation of a concordance and retrieval of the first page.

All operations are evaluated on the 130 parts corpus. The new system is evaluated in the Nymfe cluster (distributed environment), and on a single computer. The evaluation on a single computer uses a single server application and the enTenTen 2012 corpus divided into 13 parts, also referred to as 13 parts corpus. The original system is evaluated on the enTenTen 2012 corpus.

5.2.1 Computation of a Concordance

Manatee supports an asynchronous computation of a concordance. It is used by the Bonito web interface, which divides results of concordances into pages. The advantage is that a user can see the first page while the rest of the concordance is still being computed as a background task. However, many operations with a concordance processes the whole concordance, so a user must wait until the concordance finishes its computation before such operation is executed. The implemented system does not use the asynchronous evaluation and the first page of a concordance is obtained only after all servers had already computed the concordances.

The computation of a concordance is compared between the implemented system running in the Nymfe cluster, the implemented system running on a single computer, and the original system with a synchronous and asynchronous computation of a concordance.

The first pages of the evaluated concordances are obtained with the default values of the page size (20 lines) and format of the KWIC lines (40 characters before and 40 characters after the KWIC; show the word attribute, and the beginnings and ends of paragraphs; the empty **g** structure[1]; and reference to a document number).

1. Empty structure **g** denotes a **glue** (no space separation) between words, <https://www.sketchengine.co.uk/preparing-a-text-corpus-for-the-sketch-engine-overview/>

Query	Concordance size	Original s. – as. [s]	Original s. – syn. [s]	Implemented s. – Nymfe cluster [s]	Implemented s. – single comp. [s]
`[word="test.*ing"]`	721,212	12.280	14.449	2.667	30.411
`[word="work.*ing"]`	3,696,606	14.627	20.045	0.997	37.194
`[word="confus.*"]`	702,436	14.887	18.244	0.295	34.893
`[word="(?i)confus.*"]`	731,452	25.440	34.512	0.765	43.639
`[word=".*ing"]`	371,767,766	240.000	626.947	4.183	241.772
`[tag="JJ"] [lemma="plan"]`	553,724	3.183	18.214	1.330	15.743
`[lemma_lc="good"] [lc="plan"]`	20,804	6.185	6.274	0.464	18.945
`"some" [tag="NN"]`	5,107,984	3.280	36.141	1.462	22.728
`[lc=".*ing" & tag!="VVG"]`	141,174,215	61.750	229.081	5.846	281.317
`[tag="DT"] [lc=".*ly"] [lc=".*ing"] [word="[A-Z].*"]`	54,957	334.002	*more than 3600*	32.889	*more than 3600*
`[lc=".*ing" & tag="VVG"]`	231,346,778	61.100	242.514	5.013	222.768
`[tag="DT"] [lc=".*ly"] [lc=".*ing"] [word="[A-Z].*" & tag!="P.*"]`	29,053	344.571	*more than 3600*	35.443	*more than 3600*

Table 5.4: Performance of creating and retrieving the first page of a concordance compared between the implemented and original system. **Original s. – as.** denotes the original system with asynchronous evaluation. **Original s. – syn.** denotes the original system with synchronous evaluation. **Implemented s. – Nymfe cluster** denotes the original system evaluated in the Nymfe cluster. **Implemented s. – single comp.** denotes the original system evaluated on a single computer.

Computation of a concordance is evaluated with several queries. The queries are different as the queries used in other evaluations. They cover searching for regular expressions, and searching based on multiple attributes. Table 5.4 shows results of the evaluation and size of the used concordances.

The evaluation shows that the implemented system using a cluster of computers is much faster than the original system. It is faster than the both variants of the original system: synchronous and also asynchronous evaluation. The speedup is from 2.39 to 69.2.

The implemented system evaluated on a single computer with the 13 parts corpus is

always slower than the asynchronous evalation of the original system. It is faster than the synchronous evaluation for some queries, but most of the queries are evaluated slower even that the evaluation used a virtual concordance with 13 corpora. The issue can be the configuration of server's RAID 6, or the fact that the implemented system uses the server with the REST interface that is stateless and therefore, it is required to store each created concordance.

5.2.2 Frequency Distribution and the Concordance Sort Operation

Performance of the frequency distribution and the concordance sort operation are evaluated in the similar way as the computation of a concordance. The implemented system is also evaluated in the Nymfe cluster on the 130 parts corpus, and on a single computer with the 13 parts corpus. The original system is evaluated on the whole enTenTen 2012 corpus.

Result of the concordance sort operation is a sorted concordance, so the result is presented in the same way as a concordance – pages displaying the related KWIC lines. The evaluation measured the total time of creating and sorting concordance, and obtaining of the first page of the sorted concordance.

Table 5.5 and 5.6 show results of the evaluation of the sort and frequency distribution respectively. The implemented system evaluated on a cluster of computers is faster than the original system. The speedup is from 55.78 to 304.44 for the sort operation (ignoring queries that took too long to compute), and from 27.91 to 614.45 for the frequency distribution. The implemented system evaluated on a single computer is also faster than the original system. Besides the `[word="Gauss"]` query, the queries are speeded up from 2.65 to 3.46 for the sort operation, and from 2.05 to 4.99.

Query	**Original system**	**Implemented system**	
		Nymfe cluster	**single computer**
`[word="Gauss"]`	26.887	0.482	26.854
`[word="recurrence"]`	180.160	1.086	52.003
`[word="enjoyment"]`	410.078	1.347	123.934
`[word="test"]`	492.789	3.292	158.377
`[word="said"]`	266.687	4.513	100.772
`[word="a"]`	*more than 3600 s*	23.987	*more than 3600 s*
`[word="the"]`	*more than 3600 s*	54.728	*more than 3600 s*

Table 5.5: Performance of the concordance sort operation compared between the implemented and original system. The measured times include creation of concordances, sort of the concordances, and retrieving of the first pages of the sorted concordances.

Query	Original system	Implemented system	
		Nymfe cluster	single computer
`[word="Gauss"]`	17.006	0.357	12.795
`[word="recurrence"]`	159.315	0.328	31.899
`[word="enjoyment"]`	361.910	0.589	101.561
`[word="test"]`	482.941	3.667	138.392
`[word="said"]`	147.496	5.285	67.245
`[word="a"]`	576.388	15.415	136.895
`[word="the"]`	1273.009	28.858	621.957

Table 5.6: Performance of the frequency distribution compared between the implemented and original system.

5.2.3 Scalability

Amount of Data in a Cluster

Distributed operations executed on a cluster of computers are performed in parallel. However, the results are merged on a single computer which can be a bottleneck of the scalability as only one computer processes all the data and communicates with all servers.

The following comparison shows how the scalability is affected by the amount of data and number of servers in a cluster. The comparison evaluates the scalability of the frequency distribution, concordance sort operation and retrieving of the first page of a sorted concordance, and computation of a concordance and obtaining of the first page of the concordance. The operations are evaluated on the first 10, 20, 30, 40, 50, 60, and 70 servers of the Nymfe cluster. The first 10 computers of the Nymfe cluster have the first 20 parts of the 130 parts corpus. Each computer has 2 parts. The first 20 parts were copied to the next 10 computers. It means that the 20 servers produce exactly 2 times more data than the 10 servers. The first 20 parts were also copied to the other servers in the same way, so the 70 servers produce exactly 7 times more data than the 10 servers, and the 60 servers produce exactly 2 times more data than the 30 servers.

Table 5.7, 5.8, and 5.9 show the performances of the concordance sort operation and retrieving of the first page of a sorted concordance, frequency distribution, and computation of a concordance and obtaining of the first page of the concordance, respectively. The measured times are affected by other users and their processes that can consume hardware resouces, because the Nymfe cluster is not dedicated and other users could run their processes during the evaluation.

The frequency distribution and the concordance sort operation involve communication with the servers and processing of the data produced by the servers. Therefore, larger amount of data in a cluster results in more data to be received and processed by a client. The most of the results are inconclusive which is probably caused by the fact that the servers were not dedicated. For example, the frequency distribution evaluated

with the query **[word="Gauss"]** and the data from the 60 servers is slowed down 3.36 times compared to the evaluation with the data from only the 50 servers. On the other hand, the frequency distribution evaluated with the query **[word="said"]** and the data from the 60 servers is speeded-up by 1.05 times compared to the evaluation with the data from only the 50 servers. The results of the concordance sort operation are also inconsistent.

The evaluation of the computation of a concordance and obtaining of the first page of the concordance involves mostly communication with the servers because the retrieval of the first page of the concordance usually requires only one request to the first server. The computation of concordances on more servers should not affect the performance, because the computation runs in parallel and is independent. If a larger number of computers in a cluster affects the performance of the computation of a concordance, then the performance is mainly affected by the increased number of communication channels. The evaluation of the query **[word="said"]** shows that the increased number of communication channels does not affect the performance because the performance of the evaluation with the data from the 10 servers is similar to the performance with the data from the 40, 50, 60, and 70 servers, but the evaluation of the query **[word="Gauss"]** shows the opposite – the performance is worse as the number of servers is increased. The results of the computation of a concordance are also inconsistent.

Despite the fact that the servers were not dedicated, the measured data show that the performance of the implemented system is affected by the amount of data in a cluster as expected but the impact is not significant.

Query	Number of servers						
	10	**20**	**30**	**40**	**50**	**60**	**70**
[word="Gauss"]	0.293	0.570	0.828	0.813	1.020	3.239	4.089
[word="recurrence"]	0.652	0.644	0.750	0.869	1.073	1.702	2.177
[word="enjoyment"]	0.927	0.938	0.947	0.971	1.372	1.297	1.637
[word="test"]	2.105	1.929	2.054	2.058	2.657	2.366	2.483
[word="said"]	2.546	2.300	2.843	2.988	3.223	3.060	3.131
[word="a"]	22.841	23.263	23.772	24.049	24.056	26.086	30.881
[word="the"]	51.606	52.394	52.795	54.826	53.282	57.549	55.280

Table 5.7: Scalability of the distributed concordance sort operation including creation of a concordance, sort of the concordance, and retrieving of the first page of the sorted concordance.

Amount of Data on a Computer

The minimum size of a cluster is given by the maximum amount of data that can be stored on a single computer of the cluster. The more data on a server result in more disk operations and more usage of a CPU. The evaluations of the amount of data on

Query	Number of servers						
	10	**20**	**30**	**40**	**50**	**60**	**70**
`[word="Gauss"]`	0.219	0.256	0.341	0.371	0.433	1.457	1.777
`[word="recurrence"]`	0.196	0.228	0.253	0.226	0.407	0.670	0.618
`[word="enjoyment"]`	0.425	0.416	0.495	0.550	0.466	0.596	0.651
`[word="test"]`	1.597	1.541	1.654	1.694	1.679	1.880	2.278
`[word="said"]`	2.122	1.891	2.286	2.595	2.684	2.555	2.642
`[word="a"]`	8.197	8.500	10.827	10.396	11.288	11.969	12.843
`[word="the"]`	12.905	14.366	16.208	17.841	19.458	21.148	22.605

Table 5.8: Scalability of the distributed frequency distribution.

Query	Number of servers						
	10	**20**	**30**	**40**	**50**	**60**	**70**
`[word="Gauss"]`	0.115	0.111	0.172	0.200	0.260	0.579	0.940
`[word="recurrence"]`	0.181	0.115	0.174	0.236	0.219	0.218	0.293
`[word="enjoyment"]`	0.153	0.122	0.122	0.168	0.208	0.130	0.216
`[word="test"]`	0.143	0.212	0.163	0.147	0.133	0.177	0.226
`[word="said"]`	0.226	0.168	0.353	0.455	0.248	0.174	0.166
`[word="a"]`	0.911	0.859	1.386	0.995	1.096	1.049	1.050
`[word="the"]`	1.833	1.921	2.240	2.330	2.284	2.358	2.284

Table 5.9: Scalability of the distributed computation of a concordance and retrieval of the first page of the concordance.

a computer show how the performance is affected by storing more data on a single computer, and storing data in a smaller and bigger cluster.

The evaluation of storing more data on a single computer compares the performance of creating a virtual concordance containing all words from given corpora. The evaluation was executed on the Arachne server. The evaluation uses only the data of the first part of the 130 parts corpus, but the part is copied multiple times to see how the performance is affected when operations on a single computer process exactly 2, 3, 4, and 5 times more data.

The results are presented in Table 5.10. The operation is slowed down by aproximatelly 1.05 times when 2 corpora are used instead of 1 corpus. The computation with 3 corpora is slower by aproximatelly 1.26 times compared to the computation with 1 corpus. 4 corpora slowed down the operation by approximatelly 2.01 times, and 5 corpora slowed down the operation by more than 10 times. The best scalability of the computation of the virtual concordance containing all words from given corpora is achieved with 3 corpora.

The second evaluation compares two clusters with different sizes. The smaller cluster consists of 10 computers (Nymfe01–Nymfe10) and the bigger cluster consists of 20

computers (Nymfe01–Nymfe20). Both clusters provide the data of the first 20 parts of the 130 parts corpus. The bigger cluster has 1 part of the corpus on each computer. The smaller cluster is evaluated in 2 variants. The first variant uses the first 20 parts of the 130 parts corpus – each computer has 2 parts. The second variant uses merged parts. The merged parts have the same amount of data but instead of having 2 parts on each computer, there is only 1 merged part on each computer. The first variant uses parallelism to handle more data on a single computer and the second variant uses the same computations as the bigger cluster but the amount of data on a computer is larger.

The evaluation shows performance of the frequency distribution, concordance sort operation and retrieving of the first page of the sorted concordance, and creating of a concordance and retrieval of the first page of the concordance. The queries with rather larger number of results are evaluated faster when they are evaluated on the bigger cluster. The performance is the worst when the queries are evaluated on the smaller cluster with the merged parts. Most of the queries that are evaluated under 1 s have similar performance regardeless of the cluster size or number of the corpus parts on computers. It is probably caused by the overhead of network communication that outweights the evaluation of operations. The results are presented in Table 5.11, 5.12, and 5.13.

Number of corpora	**Time to create a virtual concordance containing all words [s]**	**Total size of a concordance**
1	35.359	100,174,007
2	37.386	200,348,014
3	44.520	300,522,021
4	71.250	400,696,028
5	506.260	500,870,035

Table 5.10: Evaluation of the performance affected by more data on a server.

Query	**10 servers**		**20 servers**
	Each server has 2 parts	**Each server has 1 merged part**	
[word="Gauss"]	0.168	0.143	0.187
[word="recurrence"]	0.168	0.170	0.444
[word="enjoyment"]	0.464	0.458	0.449
[word="test"]	1.683	1.322	2.292
[word="said"]	2.065	1.685	1.508
[word="a"]	8.147	10.967	7.737
[word="the"]	13.206	18.862	12.149

Table 5.11: Evaluation of the bigger and smaller cluster comparing the sort operation.

Query	10 servers		20 servers
	Each server has 2 parts	Each server has 1 merged part	
`[word="Gauss"]`	0.265	0.258	0.178
`[word="recurrence"]`	0.564	0.609	0.667
`[word="enjoyment"]`	0.897	0.963	1.043
`[word="test"]`	1.828	1.844	1.706
`[word="said"]`	2.508	2.519	2.022
`[word="a"]`	23.946	35.194	20.214
`[word="the"]`	53.313	78.451	43.915

Table 5.12: Evaluation of the bigger and smaller cluster comparing the frequency distribution.

Query	10 servers		20 servers
	Each server has 2 parts	Each server has 1 merged part	
`[word="Gauss"]`	0.189	0.298	0.137
`[word="recurrence"]`	0.139	0.105	0.104
`[word="enjoyment"]`	0.148	0.083	0.080
`[word="test"]`	0.135	0.080	0.070
`[word="said"]`	0.139	0.176	0.101
`[word="a"]`	1.026	1.333	0.788
`[word="the"]`	2.221	2.859	1.566

Table 5.13: Evaluation of the bigger and smaller cluster comparing the creating and retrieving of the first page of a concordance.

Chapter 6

Conclusions

This thesis presents parallel processing of the selected time-consuming operations of the Manatee system with large text corpora. The implemented system is much faster than the original system, which solves problems with the time-consuming operations that create unpleasant user experience and consume hardware resources.

The implemented system has the client/server architecture and implements the frequency distribution, concordance sort operation and retrieving of a page of the sorted concordance, and computation of a concordance and retrieving of the concordance page. Some features of the original system are missing but the missing functionality can be easily implemented because it is simplier or not more complex than the implemented concordance sort operation and retrieval of a page of a sorted concordance.

Evaluation of the system includes comparison with the original system and with an application implementing the MapReduce model of the frequency distribution and using the Glow framework. The Glow framework is an open-source MapReduce system. The comparison with the original system prooves that the parallelization speeds up the operations.

The comparison with the MapReduce framework shows that the MapReduce environment brings some overhead that slows down operations that do not produce large amount of results, but when an operation produce large amount of data, then the MapReduce approach has better performance than the implemented system. It is caused by the merge phase of the implemented system that is executed only on a single computer which is a bottleneck of the system.

Scalability of the implemented system is evaluated on a single computer and also on a cluster of computers. On a single computer, the system scales with a virtual concordance processing a corpus divided into parts. The number of parts should be at most equal to the number of processors of the computer. Evaluation on the Arachne server indicates that a virtual concordance scale for corpora that have at most 400 million of words.

The evaluation of the scalability of a cluster of computers shows that a bigger cluster with smaller amount of data on a computer achieve better performance than a smaller cluster with larger amount of data on a computer. The scalability is also evaluated with increasing number of servers and amount of data in a cluster. The evaluation shows that the system is affected by the amount of data in a cluster but it does not cause a significant slow down.

The implemented system, just like every distributed system, must specify format of transmitted data. Considered formats were JSON, Gob, Protocol Buffer, and Gogoproto. The formats are compared in a benchmark and the Gogoproto format was selected as the format with the best performance.

The thesis also opens a discussion if the Manatee's compression algorithm could be enhanced. The original idea was to replace the current encoding with a different one to get rid of the mapping from strings to integers, because the mapping can be inconsistent on different computers of a cluster. It turned out that all selected algorithms have slower decompression or worse compression ratio. Because of that, the mapping is preserved, but the encoding of integers could be probably replaced to achieve better performance with preserved or enhanced compression ratio. A short comparison evaluates the BP32 and FastPFOR compressions and both affect the performance of the concordance sort operation positively without significant loss of the compression ratio. A future research should be done to find an algorithm that could enhance performance and compresion ratio of the Manate's encoding.

The implemented system solves problems with the time-consuming operations and will be used in the production environment as soon as the remaining functionality of the original system will be implemented.

Bibliography

[1] ASTON, Guy; BURNARD, Lou. *The BNC handbook*. Edinburgh: Edinburgh University Press, 1998. p. 256. ISBN 0-7486-1055-3.

[2] BARONI, Marco; UEYAMA, Motoko. Building general-and special-purpose corpora by web crawling. In Proceedings of the 13th NIJL international symposium, language corpora: Their compilation and application. 2006. p. 31–40. BARONI, Marco; UEYAMA, Motoko. *Building general- and special-purpose corpora by Web crawling*

[3] BASSETT, Lindsay; *Introduction to JavaScript Object Notation: A To-the-Point Guide to JSON*. Sebastopol: O'Reilly Media, Inc., 2015. ISBN 978-1-491-929483.

[4] DEAN, Jeffrey; GHEMAWAT, Sanjay; MapReduce: Simplified Data Processing on Large Clusters. In *Commununications of the ACM*. New York: ACM, Jan. 2008. vol. 51, pp. 107–113, 7 p., ISSN 0001-0782.

[5] DONOVAN, Alan A. A., KERNIGHAN, Brian W. *The Go Programming Language*. Crawfordsville, Indiana: Addison-Wesley Professional, 2015. 400 p., ISBN 978-0-13-419044-0.

[6] ELIAS, Peter. Universal codeword sets and representations of the integers. In *Information Theory, IEEE Transactions on*. 1975, 21.2: pp. 194–203.

[7] GHEMAWAT, Sanjay; GOBIOFF, Howard; LEUNG, Shun-Tak. The Google file system. In *ACM SIGOPS operating systems review*. ACM, 2003. p. 29-43. Available from: <http://static.googleusercontent.com/media/research.google.com/sk//archive/gfs-sosp2003.pdf>.

[8] JAKUBÍČEK, Miloš; KILGARRIFF, Adam; KOVÁŘ, Vojtěch; RYCHLÝ, Pavel; SUCHOMEL, Vít . The TenTen Corpus Family. In *7th International Corpus Linguistics Conference CL 2013*. Lancaster, 2013. pp. 125–127, 3 p.

[9] JAKUBÍČEK, Miloš; RYCHLÝ, Pavel. Optimization of Regular Expression Evaluation within the Manatee Corpus Management System. In *Eighth Workshop on Recent Advances in Slavonic Natural Language Processing*. Brno: Tribun EU s.r.o., 2014. pp. 37–48, 12 p., ISSN 2336-4289.

[10] JAKUBÍČEK, Miloš; RYCHLÝ, Pavel; KILGARRIFF, Adam; MCCARTHY, Diana. Fast syntactic searching in very large corpora for many languages. In *PACLIC 24 Proceedings of the 24th Pacific Asia Conference on Language, Information and Computation*. Tokyo: Waseda University, 2010. pp. 741–747, 7 p., ISBN 978-4-905166-00-9.

[11] KERAU, Holden; KONWINSKI, Andy; WENDELL, Patrick; ZAHARIA, Matei. *Learning Spark: Lightning-Fast Big Data Analysis*. Sebastopol: O'Reilly Media, Inc., 2015. 276 p., ISBN 978-1-449-35892-4.

[12] LEMIRE, Daniel; BOYTSOV, Leonid. Decoding billions of integers per second through vectorization. In *Software: Practice and Experience*. New York: John Wiley & Sons, Ltd., 2015. pp. 1–29. DOI 10.1002/spe.2203.

[13] Lexical Computing Ltd. . *Corpus Querying: Corpus Query Language (CQL)* [online]. [visited on 20-09-2015]. Available from: <https://www.sketchengine.co.uk/corpus-querying/>.

[14] Lexical Computing Ltd. . *Frequency Page Help* [online]. [visited on 12-05-2016]. Available from: <https://www.sketchengine.co.uk/frequency-page/>.

[15] LU, Chris; *Glow: Map Reduce for Golang* [online]. 27-12-2015 [visited on 17-05-2016]. Available from: <https://blog.gopheracademy.com/advent-2015/glow-map-reduce-for-golang/>.

[16] LU, Chris; *Glow: Setup master and agents* [online]. 09-11-2015 [visited on 10-04-2016]. Available from: <https://github.com/chrislusf/glow/wiki/setup-master-and-agents/b550fa18f18334b28da03b03a8397772af336f5a>.

[17] MAHONEY, Matt. *About the Test Data* [online]. Sept. 2011 [visited on 17-05-2016]. Available from: <http://mattmahoney.net/dc/textdata.html>.

[18] MAHONEY, Matt. *Large Text Compression Benchmark* [online]. 30-03-2016 [visited on 01-04-2016] Available from: <http://mattmahoney.net/dc/text.html>.

[19] PIKE, Rob. *Gobs of data*[online]. 24-03-2011 [visited on 17-15-2016]. Available from: <https://blog.golang.org/gobs-of-data>.

[20] POMIKÁLEK, Jan; RYCHLÝ, Pavel; JAKUBÍČEK, Miloš. Building a 70 billion word corpus of English from ClueWeb. In *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)*. Istanbul, Turkey: European Language Resources Association (ELRA), 2012. pp. 502-506, 5 p. ISBN 978-2-9517408-7-7.

[21] *Powered by Apache Hadoop* [online]. 2016-04-14 12:00:19 [visited on 24-05-2016]. Available from: <http://wiki.apache.org/hadoop/PoweredBy?action=recall&rev=439>.

[22] *Protocol Buffers in Go with Gadgets: Documentation* [online]. [visited on 17-05-2016]. Available from: <http://gogo.github.io/doc/>.

[23] RÁBARA, Radoslav; RYCHLÝ, Pavel. Concurrent Processing of Text Corpus Queries. In *Ninth Workshop on Recent Advances in Slavonic Natural Language Processing*. Brno: Tribun EU, 2015. pp. 49-58, 10 p. ISBN 978-80-263-0974-1.

[24] RICHARDSON, Leonard; RUBY, Sam. *RESTful web services*. Sebastopol: O'Reilly Media, Inc., 2008. ISBN 978-0-596-52926-0.

[25] RYCHLÝ, Pavel. *Corpus Managers and their effective implementation*. Brno, 2000. XIV+128 p. PhD Thesis. Faculty of Informatics, Masaryk University.

[26] RYCHLÝ, Pavel. Manatee/Bonito - A Modular Corpus Manager. In *1st Workshop on Recent Advances in Slavonic Natural Language Processing*. Brno: Masaryk University, 2007. pp. 65-70. ISBN 978-80-210-4471-5. Available from: <http://nlp.fi.muni.cz/raslan/2007/papers/12.pdf>.

[27] SALMON, David. *Data Compression: The Complete Reference*. 4th ed. London: Springer-Verlag London Limited, 2007. pp. 176–178., ISBN 1-84628-602-6. Available from: <http://read.pudn.com/downloads167/ebook/769449/dataCompress.pdf>.

[28] WHITE, Tom; *Hadoop: The Definitive Guide, 3nd Edition*. Sebastopol: O'Reilly Media, Inc., 2012. 657 p., ISBN: 978-1-449-31152-0. Available from: <http://download.bigbata.com/ebook/oreilly/books/Hadoop.The.Definitive.Guide.3rd.Edition.May.2012.pdf>.

Appendix A

All Results of the First Text Compression Benchmark

Program name	Options	Compression time [s]	Decompression time [s]	Compressed file size [MB]	Compression ratio
LZ4-r131		0.62	0.22	77	7.17
LZ4-r131	-9	8.86	0.22	54	10.21
LZ4 [1]	-9	10.06	0.23	54	10.21
Shrinker		0.69	0.27	73	7.56
LzTurbo [2]	-32	3.94	0.33	41	13.60
LzTurbo [3]	-32 -p1	5.35	0.54	41	13.60
lzop	-9	29.66	0.57	55	10.10
eXdupe		2.09	0.58	75	7.40
lzop		0.68	0.61	79	7.05
lrzip 0.621 [-5]	-l	7.94	0.70	77	7.25
QuickLZ	-3	9.54	0.84	61	9.05
eXdupe	-x2	2.93	0.86	60	9.30
zlib-1.2.8 minigzip		9.90	0.91	48	11.63
NanoZip [4]	-cf -M1670	1.00	0.94	67	8.35
compress [5]		4.17	0.98	56	9.95
GZIP	-9	12.46	1.07	48	11.70
GZIP	-5	5.29	1.09	49	11.42
lrzip 0.621 [6]	-l -p 1	8.26	1.11	77	7.25
Info-ZIP		9.74	1.23	48	11.64
GZIP	-1	2.50	1.25	58	9.65
Info-ZIP	-9	13.96	1.25	47	11.70
zpaq-705 [-3]	-m 24	24.63	1.33	43	12.96
Info-zip	-1	2.64	1.37	58	9.65
lrzip 0.621 [-5]		75.32	1.52	35	15.79
zpaq-705 [-3]		5.23	1.55	48	11.52
QuickLZ		1.33	1.65	72	7.69
crush [7]		131.14	1.72	44	12.75

alba	-c32768	64.92	1.82	67	8.31
crush [0]	-cf	7.64	1.85	51	10.80
7-zip		108.72	2.13	34	16.22
zpaq-705 [-4]	-m 24 -threads 1	45.43	2.19	43	12.96
alba		70.60	2.30	72	7.68
XZ 5.1.0alpha		133.30	2.51	35	15.97
xwrt 32b		18.99	3.14	41	13.69
flzp		7.12	3.42	82	6.76
XZ 5.1.0alpha	-F lzma -zv0	10.83	3.55	51	10.82
NanoZip [-3]		6.44	3.72	28	19.62
bzip2	-1	12.85	5.03	41	13.47
eXdupe	-x3	9.60	5.26	40	13.89
bzip2		13.88	5.32	36	15.52
comprox [-5]	-m100 -b1000 -f	224.23	5.52	35	16.05
bzip2	-9	13.73	5.59	36	15.52
comprox [-5]	-m100 -b1000	67.98	5.82	35	15.86
srank		6.56	5.93	62	9.00
srank	-C8	6.68	6.05	58	2.43
comprox [2]	-m40 -b16	28.81	6.09	36	3.86
srank	-C1	6.78	6.22	73	1.92

Table A.1: All results of the first text compression benchmark.

Appendix B

Data and Results of The Second Text Compression Benchmark

Program name	Options	Compression time [s]	Decompression time [s]	Compressed file size [MB]	Compression ratio
LZ4-r131	-9	36.29	0.95	223.97	2.48
LZ4-r131		4.00	0.99	317.80	1.74
LzTurbo [-5]	-32	20.02	1.60	167.30	3.32
lzop	-9	118.41	2.24	225.59	2.46
LzTurbo [-4]	-32 -p1	22.60	2.26	167.30	3.32
Shrinker		7.28	2.27	299.00	1.86
zlib-1.2.8 minigzip		40.21	3.69	196.00	2.83
compress [-2]		17.81	3.86	228.12	2.43
Tornado 0.6	-1	4.87	4.25	332.01	1.67
GZIP	-5	23.47	4.49	199.37	2.79
GZIP	-9	51.94	4.70	194.92	2.85
zpaq 705 [-3]		95.58	5.22	176.93	3.14
Info-zip	-9	57.01	5.22	194.92	2.85
Info-zip		43.69	5.31	195.90	2.84
Tornado 0.6	-5	17.21	5.66	190.16	2.92
lrzip 0.621 [-5]		276.99	5.744	144.87	3.84

Table B.1: Results of the second text compression benchmark: compression of the word attribute.

-6. source: <https://github.com/Cyan4973/lz4> (last commit: d86dc91)
-5. enabled multithreading
-4. single-threaded
-3. uses 2 threads
-2. Unix command
-1. multithreaded compression and single-threaded decompression
0. http://compressme.net

Program name	Options	Compression time [s]	Decompression time [s]	Compressed file size [MB]	Compression ratio
LZ4-r131		5.10	1.41	346.64	2.17
LzTurbo [-5]	-32	17.50	1.49	179.11	4.20
LZ4-r131	-9	67.27	1.54	237.56	3.17
LzTurbo [-4]	-32	22.50	2.36	179.11	4.20
lzop	-9	260.78	2.60	248.10	3.03
Shrinker		9.32	2.90	335.64	2.24
zlib-1.2.8 minigzip		46.79	4.56	212.08	3.55
compress [-2]		22.83	5.37	245.48	3.07
GZIP	-9	114.98	5.48	208.61	3.60
GZIP	-5	25.03	5.51	217.95	3.45
Info-zip	-9	126.40	6.72	208.61	3.61
zpaq 705 [-3]		113.32	7.09	183.01	4.11
Info-zip		47.32	7.27	211.94	3.55
lrzip 0.621 [-5]		334.68	7.89	149.20	5.04

Table B.2: Results of the second text compression benchmark: compression of the lempos attribute.

Program name	Options	Compression time [s]	Decompression time [s]	Compressed file size [MB]	Compression ratio
LZ4-r131		2.39	0.73	177.63	2.49
LzTurbo [2]	-32	4.41	0.79	87.79	5.04
LZ4-r131	-9	87.07	0.79	99.78	4.43
LzTurbo [-4]	-32	6.50	1.19	87.79	5.04
lzop	-9	318.02	1.25	102.42	4.32
Shrinker		4.36	2.08	181.74	2.43
zlib-1.2.8 minigzip		35.00	2.12	83.77	5.28
GZIP	-9	229.72	2.60	78.57	5.63
Info-zip		35.00	2.99	83.73	5.29
zpaq 705 [-3]		48.38	3.15	78.17	5.66
GZIP	-5	14.80	3.19	91.39	4.84
Info-zip	-9	186.70	3.85	78.57	5.63
lrzip 0.621 [-5]		241.71	4.15	66.75	6.63
compress [-2]		10.55	4.18	78.22	5.66

Table B.3: Results of the second text compression benchmark: compression of the ambtag attribute.

File	Word attribute	Lempos attribute	Ambtag attribute
Compressed text	200 MB	192 MB	95 MB
Lexicon	7.2 MB	8 MB	484 B
Index of lexicon	3 MB	2.8 MB	364 B

Table B.4: File sizes of the compressed BNC's word, lempos, and ambtag attribute.

Appendix C

Results of the Virtual Concordance Benchmark

Query	#P	Number of parts							
		2	**3**	**4**	**5**	**6**	**7**	**8**	**9**
`[word="Gauss"]`	1	0.019	0.021	0.021	0.020	0.017	0.014	0.015	0.018
	8	0.011	0.011	0.008	0.010	0.006	0.005	0.006	0.007
`[word="recurrence"]`	1	0.120	0.113	0.119	0.106	0.102	0.096	0.099	0.100
	8	0.077	0.058	0.052	0.049	0.039	0.033	0.034	0.035
`[word="enjoyment"]`	1	0.281	0.263	0.265	0.261	0.251	0.268	0.253	0.261
	8	0.150	0.111	0.095	0.080	0.064	0.058	0.047	0.061
`[word="test"]`	1	1.368	1.412	1.372	1.403	1.406	1.395	1.408	1.425
	8	0.674	0.489	0.384	0.306	0.267	0.246	0.233	0.231
`[word="said"]`	1	6.423	6.487	6.486	6.457	6.441	6.491	6.394	6.465
	8	3.039	2.147	1.880	1.600	1.263	1.230	1.206	1.136
`[word="a"]`	1	29.055	28.490	28.759	28.104	27.892	27.722	28.435	27.794
	8	13.971	10.014	8.498	7.957	7.682	7.294	7.067	6.941

Table C.1: Performance of the sort operation executed with the virtual concordance compared between the BNC corpus divided to various number of parts. **#P** denotes number of available processors.

Query	#P	Number of parts							
		2	**3**	**4**	**5**	**6**	**7**	**8**	**9**
`[word="Gauss"]`	1	0.011	0.011	0.011	0.012	0.010	0.010	0.010	0.010
	8	0.006	0.005	0.004	0.006	0.004	0.004	0.003	0.003
`[word="recurrence"]`	1	0.089	0.090	0.087	0.082	0.077	0.076	0.077	0.077
	8	0.057	0.042	0.039	0.038	0.031	0.026	0.024	0.024
`[word="enjoyment"]`	1	0.241	0.236	0.234	0.229	0.221	0.217	0.215	0.222
	8	0.139	0.101	0.090	0.070	0.053	0.049	0.038	0.043
`[word="test"]`	1	1.213	1.187	1.196	1.200	1.203	1.209	1.213	1.198
	8	0.626	0.458	0.360	0.283	0.240	0.222	0.206	0.211
`[word="said"]`	1	4.862	4.925	4.903	4.900	4.877	4.921	4.941	4.943
	8	2.415	1.758	1.429	1.137	0.907	0.853	0.796	0.802
`[word="a"]`	1	10.255	10.418	10.662	10.880	10.912	10.933	10.965	11.005
	8	5.010	3.594	3.103	2.784	2.538	2.329	2.290	2.280

Table C.2: Performance of the frequency distribution executed with the virtual concordance compared between the BNC corpus divided to various number of parts. **#P** denotes number of available processors.

Appendix D

Results of the Split Concordance Benchmark

Query	#P	Number of splits							
		2	**3**	**4**	**5**	**6**	**7**	**8**	**9**
`[word="Gauss"]`	1	0.016	0.016	0.016	0.017	0.017	0.017	0.018	0.020
	8	0.010	0.007	0.006	0.005	0.005	0.004	0.004	0.004
`[word="recurrence"]`	1	0.116	0.118	0.119	0.122	0.120	0.121	0.119	0.120
	8	0.083	0.064	0.049	0.039	0.036	0.034	0.030	0.031
`[word="enjoyment"]`	1	0.268	0.269	0.276	0.275	0.273	0.279	0.281	0.271
	8	0.161	0.122	0.091	0.074	0.067	0.063	0.057	0.071
`[word="test"]`	1	1.414	1.433	1.402	1.439	1.416	1.432	1.429	1.413
	8	0.706	0.496	0.402	0.342	0.292	0.268	0.260	0.252
`[word="said"]`	1	6.397	6.407	6.463	6.420	6.426	6.363	6.339	6.383
	8	3.302	2.435	2.038	1.753	1.520	1.407	1.300	1.281
`[word="a"]`	1	29.768	28.688	28.398	28.336	28.740	28.367	28.727	28.728
	8	17.296	11.482	9.445	9.367	8.526	7.911	7.606	7.638

Table D.1: Performance of the sort operation executed with the split and various number of splits. **#P** denotes number of available processors.

Query	#P	Number of splits							
		2	**3**	**4**	**5**	**6**	**7**	**8**	**9**
`[word="Gauss"]`	1	0.011	0.010	0.010	0.011	0.012	0.011	0.012	0.012
	8	0.007	0.005	0.004	0.003	0.003	0.003	0.003	0.003
`[word="recurrence"]`	1	0.094	0.096	0.098	0.096	0.092	0.094	0.099	0.093
	8	0.067	0.055	0.039	0.034	0.032	0.029	0.027	0.024
`[word="enjoyment"]`	1	0.256	0.252	0.264	0.247	0.251	0.248	0.264	0.246
	8	0.149	0.109	0.082	0.069	0.063	0.055	0.051	0.066
`[word="test"]`	1	1.208	1.209	1.226	1.253	1.239	1.110	1.184	1.139
	8	0.645	0.476	0.375	0.322	0.271	0.252	0.237	0.240
`[word="said"]`	1	4.925	4.837	4.876	4.826	4.930	4.965	4.972	5.024
	8	2.759	2.057	1.701	1.433	1.210	1.114	0.985	0.979
`[word="a"]`	1	9.463	9.616	9.605	9.717	9.770	9.785	9.824	9.796
	8	7.331	4.396	3.709	3.802	3.517	3.176	2.910	2.876

Table D.2: Performance of the frequency distribution executed with the split and various number of splits. **#P** denotes number of available processors.

Appendix E

Results of the Glow in Standalone Mode Benchmark

Query	#P	Number of parts							
		2	**3**	**4**	**5**	**6**	**7**	**8**	**9**
`[word="Gauss"]`	1	0.083	0.110	0.124	0.152	0.171	0.192	0.181	0.217
	8	0.071	0.089	0.108	0.128	0.141	0.156	0.163	0.189
`[word="recurrence"]`	1	0.158	0.172	0.202	0.219	0.243	0.269	0.280	0.314
	8	0.096	0.100	0.115	0.128	0.144	0.159	0.165	0.198
`[word="enjoyment"]`	1	0.298	0.324	0.351	0.362	0.377	0.405	0.436	0.458
	8	0.176	0.153	0.154	0.154	0.168	0.167	0.180	0.196
`[word="test"]`	1	1.257	1.281	1.331	1.355	1.370	1.387	1.434	1.473
	8	0.669	0.515	0.418	0.362	0.346	0.333	0.320	0.340
`[word="said"]`	1	5.002	5.137	5.215	5.320	5.387	5.440	5.503	5.618
	8	2.442	1.812	1.483	1.228	1.008	0.963	0.918	0.921
`[word="a"]`	1	10.451	10.775	10.943	11.192	11.257	11.209	11.302	11.380
	8	5.053	3.533	3.097	2.824	2.583	2.429	2.377	2.367

Table E.1: Performance of the frequency distribution executed with the program using Glow compared between the BNC corpus divided to various number of parts. **#P** denotes number of available processors.

Appendix F

Results of Benchmark of the Data Transmission with Different Formats

Gob is evaluated in 2 variants: Gob* and Gob**. Gob* creates a new instance of Encoder and Decoder each time a data type is encoded and decoded. Gob** uses one shared instance of Encoder and Decoder. The sizes of the encoded data by Gob** are denoted with ***, because they were measured after the data types had been already compiled.

Format	**Average time of the sort data transmission [s]**	**Size of the encoded data**		
		Average	**Minimum**	**Maximum**
JSON	7.550	9.550 KB	3.260 KB	15.841 KB
Gob*	5.026	5.265 KB	2.032 KB	8.499 KB
Gob**	3.205	***4.956 KB	***1.723 KB	***8.499 KB
Protocol Buffer	2.305	5.061 KB	1.755 KB	8.368 KB
Gogoproto	1.557	5.061 KB	1.755 KB	8.368 KB

Table F.1: Benchmark of the sort data transmission with the query `[word="Gauss"]` and 10,000 repetitions.

Format	**Average time of the sort data transmission [s]**	**Size of the encoded data**		
		Average	**Minimum**	**Maximum**
JSON	6.106	15.762 KB	15.405 KB	16.872 KB
Gob*	3.669	8.873 KB	8.679 KB	9.092 KB
Gob**	2.647	***8.564 KB	***8.370 KB	***8.783 KB
Protocol Buffer	1.895	8.667 KB	8.451 KB	8.991 KB
Gogoproto	1.259	8.667 KB	8.451 KB	8.991 KB

Table F.2: Benchmark of the sort data transmission with the query `[word="recurrence"]` and 1,000 repetitions.

Format	Average time of the sort data transmission [s]	Size of the encoded data		
		Average	Minimum	Maximum
JSON	12.391	15.828 KB	15.056 KB	16.690 KB
Gob*	7.586	8.772 KB	8.444 KB	8.933 KB
Gob**	5.429	***8.463 KB	***8.135 KB	***8.624 KB
Protocol Buffer	3.972	8.593 KB	8.226 KB	8.852 KB
Gogoproto	2.665	8.593 KB	8.226 KB	8.852 KB

Table F.3: Benchmark of the sort data transmission with the query `[word="enjoyment"]` and 1,000 repetitions.

Format	Average time of the sort data transmission [s]	Size of the encoded data		
		Average	Minimum	Maximum
JSON	14.580	15.863 KB	13.401 KB	18.150 KB
Gob*	9.667	8.645 KB	7.066 KB	9.177 KB
Gob**	6.835	***8.336 KB	***6.757 KB	***8.868 KB
Protocol Buffer	5.465	8.473 KB	6.914 KB	9.188 KB
Gogoproto	3.726	8.473 KB	6.914 KB	9.188 KB

Table F.4: Benchmark of the sort data transmission with the query `[word="test"]` and 100 repetitions.

Format	Average time of the sort data transmission [s]	Size of the encoded data		
		Average	Minimum	Maximum
JSON	2.906	16.720 KB	5.658 KB	19.670 KB
Gob*	1.867	8.685 KB	3.155 KB	9.423 KB
Gob**	1.357	***8.376 KB	***2.846 KB	***9.114 KB
Protocol Buffer	1.172	8.592 KB	2.916 KB	9.650 KB
Gogoproto	0.795	8.592 KB	2.916 KB	9.650 KB

Table F.5: Benchmark of the sort data transmission with the query `[word="said"]` and 1 repetition.

Format	Average time of the frequency data transmission [s]	Size of the encoded data
JSON	15.748	2.987 MB
Gob*	6.030	1.069 MB
Gob**	5.227	***1.069 MB
Protocol Buffer	5.848	1.141 MB
Gogoproto	3.940	1.141 MB

Table F.6: Benchmark of the frequency distribution data transmission with the query `[word="a"]` and 100 repetition.

Format	Average time of the frequency data transmission [s]	Size of the encoded data
JSON	13.844	270.950 KB
Gob*	4.651	86.102 KB
Gob**	4.376	***86.016 KB
Protocol Buffer	3.893	93.18 KB
Gogoproto	1.833	93.18 KB

Table F.7: Benchmark of the frequency distribution data transmission with the query `[word="said"]` and 1,000 repetition.

Format	Average time of the frequency data transmission [s]	Size of the encoded data
JSON	23.611	46.400 KB
Gob*	8.207	14.760 KB
Gob**	7.498	***14.674 KB
Protocol Buffer	5.690	15.900 KB
Gogoproto	2.036	15.900 KB

Table F.8: Benchmark of the frequency distribution data transmission with the query `[word="test"]` and 10,000 repetition.

Format	Average time of the frequency data transmission [s]	Size of the encoded data
JSON	19.212	3.574 KB
Gob*	10.744	1.057 KB
Gob**	6.341	***0.971 KB
Protocol Buffer	4.810	1.064 KB
Gogoproto	1.698	1.064 KB

Table F.9: Benchmark of the frequency distribution data transmission with the query `[word="enjoyment"]` and 100,000 repetition.

Format	Average time of the frequency data transmission [s]	Size of the encoded data
JSON	12.771	2.379 KB
Gob*	8.625	763 B
Gob**	4.252	***677 B
Protocol Buffer	3.253	737 B
Gogoproto	1.243	737 B

Table F.10: Benchmark of the frequency distribution data transmission with the query **`[word="recurrence"]`** and 100,000 repetition.

Format	Average time of the frequency data transmission [s]	Size of the encoded data
JSON	28.706	501 B
Gob*	53.529	216 B
Gob**	10.674	***130 B
Protocol Buffer	8.418	139 B
Gogoproto	2.864	139 B

Table F.11: Benchmark of the frequency distribution data transmission with the query **`[word="Gauss"]`** and 1,000,000 repetition.

Appendix G

List of Attachments

G.1 manatee-go-dist.zip

The attached zip file contains all source code of the implemented system. The client of the implemented system is in the directory **`src/manatee/cmd/concdistclient`** and the server in the directory **`src/manatee/cmd/concdistserver`**.

The program implementing the frequency distribution and using the modified Glow framework can be found in the directory **`src/manatee/cmd/concdistglow`**. The framework can be found in the directory **`src/github.com/chrislusf/glow`**.

G.1.1 Installation

- Install Go (Golang)[1] – at least version 1.5 (this thesis uses go1.5.1 darwin/amd64)
- Install the dependencies of the implemented system:

```
go get golang.org/x/text/language
go get github.com/gogo/protobuf/proto
```

- Install the dependencies of the Glow framework:

```
go get github.com/Redundancy/go-sync
go get github.com/psilva261/timsort
go get github.com/golang/protobuf/proto
go get gopkg.in/alecthomas/kingpin.v2
```

- Install the programs:

```
go install manatee/cmd/concdistserver
go install manatee/cmd/concdistclient
go install manatee/cmd/concdistglow
```

- Install the modified Glow framework:

```
go install github.com/chrislusf/glow
```

1. <https://golang.org>

G.1.2 How to run the programs

Server

It is very simple to start the server: just run the program. The server runs on the port 8080, but the port number can be changed by the command line argument **-port PORT_NUMBER**.

```
./bin/concdistserver
```

Client

To start the client, a configuration file must be defined. The configuration file is described in the section 4.2. The file contains name of a corpus followed by a list of servers with names of the corpus parts located on the server. The corpus parts on the server are described by their common prefix, and range of suffix numbers. For example, consider that there are the following corpus parts on the localhost server: **corpus01**, **corpus02**, and
corpus03. The listing G.1 shows the required configuration file of the localhost server running on the default port (8080) and with the corpus parts located in the **/tmp** directory.

Listing G.1: Sample configuration file with one server

```
corpus
http://localhost:8080,/tmp/corpus,1,3,2
```

The number after the range of suffix numbers denotes the size of the suffix. Consider that there is another localhost server running on the port 8090 and the **/tmp/second-server** directory contains the following corpus parts: **corpus004** and **corpus005**. The listing G.2 shows the required configuration file of the two localhost servers.

Listing G.2: Sample configuration file with two servers

```
corpus
http://localhost:8080,/tmp/corpus,1,3,2
http://localhost:8090,/tmp/secondserver/corpus,4,5,3
```

The client runs with the configuration file, concordance name, name of the required operation, corpus name, query, and criteria (criteria can be empty or omitted when the **CONC** operation is specified – create a concordance and get the first concordance page). The following code shows examples of invoking the client:

```
./bin/concdistclient configFileName concordanceName SORT corpus '[word="gang"]'\
'word/i 1>0~3>0'
./bin/concdistclient configFileName concordanceName FREQ corpus '[word="of"]' 'word 1>0'
./bin/concdistclient configFileName concordanceName CONC corpus '[word="four"]'
```

Program using the Glow framework

The program using the Glow framework implements the MapReduce model of the Frequency distribution as described in the section 3.2.1. It is necessary to setup the Glow cluster as described in [15], but the agents must have specified the command line argument **`--resources LIST_OF_CORPUS_PARTS`**[2]. The program requires the query, criteria, and list of corpus parts:

```
./bin/concdistglow -glow -glow.leader 'localhost:8930' '[word="test"]' 'word 1>0' \
corpus001 corpus002 corpus003
```

2. a comma separated list of the corpus parts