NBLucene: Flexible and Efficient Open Source Search Engine

Zhaohua Zhang, Benjun Ye, Jiayi Huang, Rebecca Stones, Gang Wang(✉), and Xiaoguang Liu(✉)

College of Computer and Control Engineering, Nankai University, Tianjin, China
{zhangzhaohua,yebj,huangjy,becky,wgzwp,liuxg}@nbjl.nankai.edu.cn

Abstract. The most popular open source projects for text searching have been designed to support many features. These projects are well-written in Java for cross-platform using. But when conducting research, the execution efficiency of program should be more essential, which is a problem for applications written in Java. It is also difficult for Java to use parallel mechanisms in the modern computer system like SIMD and GPUs. To this end, we expand an open source text searching project written in C++ for research purpose.

Our approach is to define a flexible and efficient search engine architecture which consists of extensible application programming interfaces. We aim to provide a flexible architecture to enable researchers to readily implement and modify search engine algorithms and strategies. Moreover, we integrate one generic mathematical encoding library which can be used to compress inverted index. We also implement an integral framework for result summarization, including snippet generation and cache strategies. Experiment results show that the new architecture makes a significant improvement versus original work.

1 Introduction

Due to the complex requirements, more and more open source projects for text searching have been well designed [11]. These projects support full-featured methods including text indexing, query processing and result presentation. For cross-platform and high performance purposes, most of them were written in Java, such as Lucene[1] and Nutch[2]. Because of their excellent design, these projects have been widely used in academic and commercial situations.

When conducting research, however, the projects which are written in Java face efficiency problems. Java programs have to been executed on a JVM (Java Virtual Machine), which allows application programs to be run on any platform without having to be rewritten or recompiled for each individual platform. This runtime environment does not directly support some parallelism in modern computer system, like SIMD (Single Instruction, Multiple Data) and GPUs. Both of

[1] https://lucene.apache.org/.
[2] http://nutch.apache.org/.

B. Cui et al. (Eds.): WAIM 2016, Part I, LNCS 9658, pp. 504–516, 2016.
DOI: 10.1007/978-3-319-39937-9_39

them have been utilized to boost query processing in [3,8,10,13,14,18], typically written in C/C++. The other problem of using Java is memory management. As the garbage collection and memory reallocation is fully controlled by the JVM, it is difficult for programmers to design the memory layout and control reallocation as needed. In some specific situations, such as cache strategy experimentation, it is a serious problem that the memory space can not be fully controlled by designers.

This paper is instead dedicated to design an efficient open source library for text searching written in C++. The core library should be designed to have good flexibility, which can be expanded for different specific algorithms and strategies expediently. As a result, we make a choice to expand the CLucene[3], which is a C++ migration of Lucene, with our experimental interfaces. We call this expansion of CLucene *NBLucene*.

The new experimental interfaces in NBLucene involve different stages during text searching, including text preprocessing, index compression, query processing and result summarization. In text preprocessing, we provide full support to analyze TREC web format and Web Archive format, which is the standard format for GOV2 and ClueWeb09 web collection, respectively. This work has not been implemented in the original CLucene project. For index compression and query processing, we implement several typical mathematical encoding methods with both scalar and SIMD versions. These methods can be used to compress posting lists and position information in the inverted index. Besides these, we also design a full-featured cache framework for result summarization, including snippet generation and cache strategies. All these interfaces can be easily expanded for other specific methods as needed. To the best of our knowledge, there is no previous open source project that provides all of the above features.

2 Related Work

2.1 Open Source Search Engines

Many text search engines have been made open source for researching and commercial use. Table 1 concludes the most popular projects for full-text searching. Lucene has become a top-level Apache project since 2005 and has been extended to a series of Lucence-based search engine, e.g. Nutch and Solr[4]. Lucene is not a complete search engine, but provides the core API library for full-text searching. CLucene is an existing port of Java Lucene written in C++. The latest version is converted from Lucene 2.3.2 and has not updated since March, 2011. Because our previous work has been integrated in Java Lucene, we refer to expand the CLucene as our baseline. Besides CLucene, there are also many other open source search engines written in C/C++, e.g. Indri [15] and Zettair [22].

[3] http://clucene.sourceforge.net/.
[4] http://lucene.apache.org/solr/.

Table 1. Comparison of the related open source search engines

Project	License	Language	Latest update
CLucene	Apache LGPLv2	C++	March, 2011
Galago	BSD	Java	January, 2016
Indri	BSD	C/C++	January, 2016
Lucene	Apache	Java	January, 2016
Nutch	Apache	Java	January, 2016
Solr	Apache	Java	January, 2016
Sphinx	GPLv2	C++	September, 2015
Terrier	MPL	Java	April, 2015
Zettair	BSD	C	March, 2009

2.2 Index Compression

In most search engines, an inverted index is the central data structure which maps terms to the documents that contain them. Given a collection of N documents, each document is represented by a unique document identifier, *docID*, between 0 and $N-1$. An inverted index contains posting lists for all distinct terms in the collection. As the posting lists often take large fraction of storage, many previous studies focus on index compression techniques [1,2,5,19,20].

Besides compression ratio, decompression speed is also a crucial indicator since the compressed posting lists relative to a query have to be completely or partially decompressed during query processing. By using SIMD instructions, several papers have reported significant improvement on decoding speed. Willhalm [18] proposed a SIMD approach to execute the vectorized value decompression with very short latency. Stepanov [14] concentrated on their SIMD-based method, called *varint-G8IU*, and outperformed the classic variable byte coding methods. Zhang [21] reported a compression framework with a novel storage layout format and made very competitive performance. Lemire [9] found that compressing integers in large blocks of integers with minimal branching would be faster than previous algorithms.

2.3 Snippet Generation

Nowadays, most search engines will return a document summarization with each top-ranked result. These summarized text fragments will help users to judge the relevance before clicking the link to get full document. Previous work on document summarization can be categorized as either query-dependent summarization or query-independent summarization. In this paper, we focus on query-dependent method, which is known as snippet generation, i.e. to rank and select most relative text fragments for each query. This method has been studied in respective papers [4,16,17].

3 Architecture

CLucene already provides a basic API for text indexing and ranked search for common query models. By keeping the standard modular architecture, we define a series of modular interfaces which would be essential for relevant experiments. These scalable modules involve the different stages in the text searching, which consist of four major components: text preprocess, index compression, query process and result summarization. Figure 1 illustrates the overall architecture of NBLucene. The rectangles with grey background are the modular interfaces we have added to the original CLuence.

The text preprocess module would parse origin HTML pages and generate formatted documents for *Indexer*. With these documents, the index compression module will build an inverted index and use *Encoder* for compression. During query processing, *Decoder* will be called to decompress posting list blocks and corresponding position lists. For each top ranked document, the result summarization module will return the snippet directly when it is in the cache. Otherwise it will generate snippet and update cache as needed.

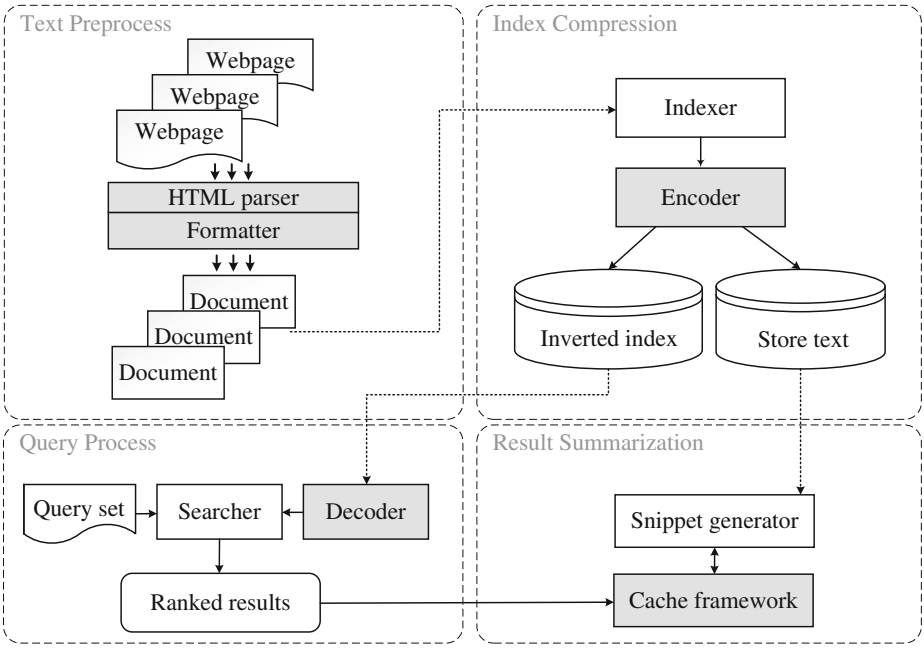

Fig. 1. Search engine architecture of NBLucene

3.1 Text Preprocess

Since CLucene does not provide any toolkit for HTML parsing, we have added an independent module for text preprocessing. The specific method for a given text collection file type needs to be implemented to compose independent documents. Each document has been stripped of HTML tokens and JavaScript, and only keeps the readable content of the original webpage. We implement the function for analysis of the TREC web format and Web Archive format corpus, and it is written in Java with JSoup[5].

In our experiments, we define four basic field types for webpage: URL, title, anchor text and content. In the text preprocess module, *Formatter* can split different fields into fixed positions. For instance, URL text, title words and anchor words are stored on the first three lines for each document, respectively. The content text will be kept next. For word stemming and stop words filtering, we would leave the option for users to determine whether they need them.

3.2 Index Compression

After index building, applications based on NBLucene may need to select different mathematical encoding methods to yield different trade-offs between space and time. The basic list compression method in CLucene is VByte [6]. We integrate a standard integer list compression library[6] in NBLucene to support other mathematical encoding algorithms. The methods of compression are implemented from unified interfaces. As instances, we also implement several common encoding and decoding methods, including both scalar and SIMD versions. For query processing efficiency, NBLucene utilizes skip-table to support random access on posting lists and position lists. For compression, inverted lists are also compressed in blocks consisting of a fixed number of postings in order to align to skip-table entries.

3.3 Query Processing and Result Summarization

To support result summarization, we integrate the standard query processing with an extensible snippet generation and cache framework. After result ranking, the most related documents, called Top-K results, will be returned. A most widely used relevance score, i.e. BM25 [12], is used in CLucene. Since CLucene has already implemented a snippet generation method which is called *Highlighter* in Lucene, we use it directly to extract the most relative text fragments for each result document. To improve query processing throughput, we utilize caching to store snippet results for future duplicate queries. In our experiment, we implement two basic cache types: results cache and snippet cache. The results cache stores queries and the corresponding result snippet for each Top-K document, and the snippet cache keeps snippet for each pair of query and result document.

[5] http://jsoup.org/.
[6] https://github.com/lemire/FastPFor.

We also implement two typical cache strategies, i.e. LRU and LFU. The caching framework is also easy to be extended for other types of cache, e.g. document cache and fragment cache, and other cache strategies.

4 Index Encoding

4.1 Posting and Position List Compression

In CLucene, the inverted index stores the relationship information between terms and documents. As CLucene supports incremental indexing, the full index is often composed of multiple sub-indexes, which are also called segments. In each segment, the index is organized into files storing distinct content. Among these files, the *term frequency file* stores the posting lists. While the *term proximity file* stores the position information of terms in each document. In this section, we will focus on these two types of files.

The term frequency file contains the lists of postings that each list is corresponding to one specific term. Each term list is composed of two parts, the posting list itself and the skip table. Figure 2 shows the layout of term frequency files. Each posting list is composed of postings and each posting is made up of a docID and the term frequency. Every docID and its term frequency will follow one special rule: if the term frequency is equal to one, its value will be omitted. In this case, the docID will be shifted left by one bit and the least significant bit will be set to 1. Otherwise, term frequency is greater than one, and we store it next to its docID. The docID will be also shifted left by one bit, but leave the least significant bit to be 0. Whether or not the frequency values are omitted, a compression method will encode them consistently.

After encoding, the term frequency file almost keeps the same format. Each term list still consists of a posting list and a skip table. In CLucene, the option *SkipInterval* is the interval between consecutive entries of the skip table. In other words, it is equal to the size of one block in the posting list. In our implementation, we compress each block independently. The first docID of each block will be stored in the corresponding skip table entry. After compression, the offset of each code words block will be updated in skip table.

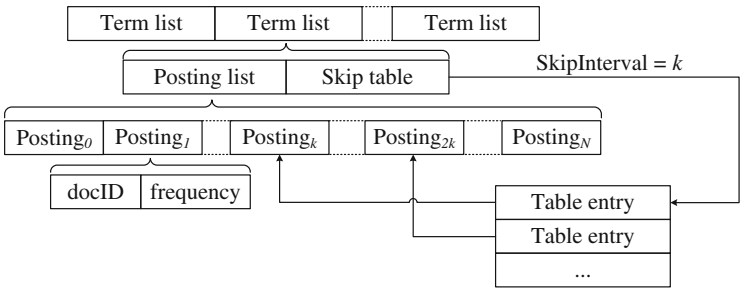

Fig. 2. Original file format of the term frequency file

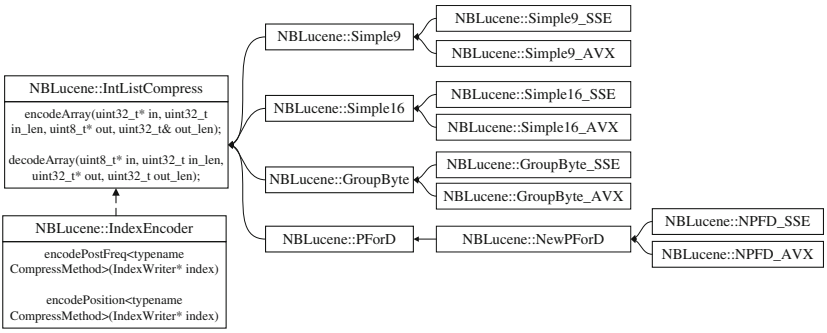

Fig. 3. Class inheritance and interfaces of list compression

Similar to the term frequency file, the term proximity file stores the lists of positions in which the term occurs in documents. We compress the position lists within each block of corresponding posting list. After compression, the offset of position list code words in the skip table entries will also be updated.

Figure 3 shows the interfaces and the class hierarchy of list compression. The base class of list compressors is *IntListCompress*, which has two main interfaces: *encodeArray* and *decodeArray*. The derived classes, such as Simple9 in the figure, or other methods designed by the users, have to be implemented for the specific method to encode original integer lists and decode code words, respectively. For encodeArray, there are four arguments: the origin integer list *in* with its length *in_len* and the output byte sequence *out* with its length *out_len*, while for decodeArray the arguments are similar. *IndexEncoder* is a C++ template which based on the derived class of IntListCompress, and it makes use of one specific encoding method to compress posting lists and position information.

4.2 Encoding Algorithm

Due to the lack of generic mathematical encoding methods in CLucene, we implement several typical algorithms to compress the index. In contrast to other previous mathematical encoding libraries, we have also provided relative SIMD implementation for different decoding algorithm, including Simple 9 [2], Simple 16 [20], NewPForD [19] and GroupByte [7]. We make use of the FastPFor and varint-G8IU as SIMD implementations for NewPForD and GroupByte, as given in [9] and [14], respectively.

For Simple 9 and Simple 16, we design a parallel decoding method. To illustrate, Fig. 4 shows the decoding procedure for an example. Suppose there are four consecutive integers b_0, b_1, b_2, b_3 which are compressed into one Simple 9 code word. That means each integer can be represented within seven bits. The parallel decoding method will first make four copies C_0, C_1, C_2, C_3 of the code word in a SIMD register as Fig. 4(a) shows. In Fig. 4(b), by utilizing parallel instructions, we can shift b_0, b_1, b_2, b_3 to the last position in C_0, C_1, C_2, C_3, respectively. Finally, we use the parallel bitwise AND instruction to "clear" other bits

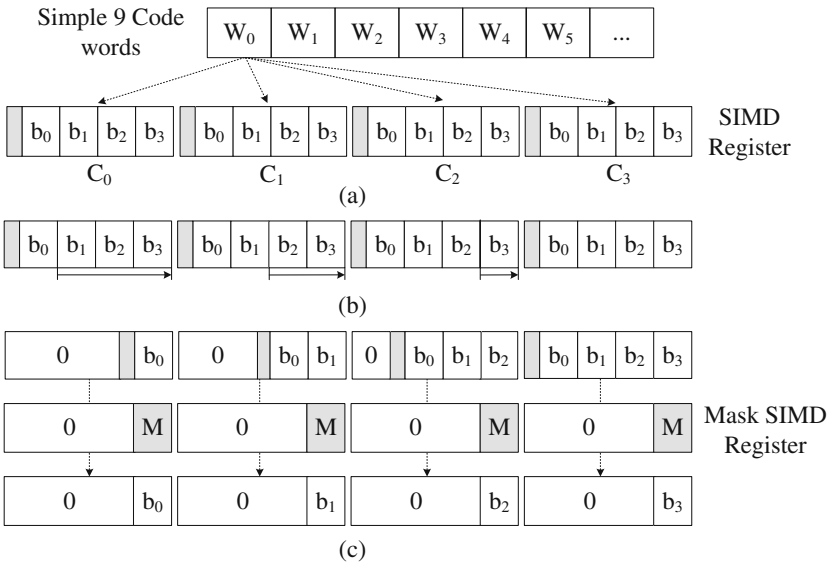

Fig. 4. Decoding procedure of Simple 9 in SIMD scheme

in C_0, C_1, C_2, C_3 leaving b_0, b_1, b_2, b_3. In Fig. 4(c), each rectangle with a grey background in Mask SIMD Register contains seven consecutive bits 1. The content in the last register is the final result. In our experiment, we implement these SIMD algorithms on an Intel platform with SSE and AVX instructions.

5 Cache Framework

CLucene provides one straightforward snippet generation method which we extend with cache strategies. We implement the cache framework and provide the interfaces for different types of cache. The kernel data structure of a cache is a *cache table*, and an entry in this table is a *cache node*, which is defined by a key-value pair. For instance, the key of a cache node in the results cache is query string and the value is the arrays of snippets of the top-ranked documents. When receiving a new query, the search engine will first lookup the cache table to find the results. If it succeeds, the result will be returned directly. Otherwise, it will search the top-ranked documents and generate snippets from them. According to its cache policy, the cache framework will decide whether or not to store a new snippet result, and which cache entries are replaced if available space is insufficient.

In addition to the kernel data structure and standard operations, we also define the interface for the cache replacement strategy for users to implement their own cache algorithms. The specific cache strategy is independent from cache type. We implement some basic cache replacement algorithms, including LRU and LFU.

1. LRU (Least Recently Used). We keep a list of pointers to record each cache node in the cache table. When a new cache node is inserted, or an existing cache node is hit, the corresponding element in the list will be moved to the head. When cache replacement occurs, the cache nodes will be removed from the tail of the list in reverse order. So the least recently used cache nodes will be evicted.
2. LFU (Least Frequently Used). For each cache node, we use a counter to record the access frequency. The frequency will increase if the cache node is hit. There is a *MinHeap* to maintain the frequency of all cache nodes. When cache replacement happens, the top element in the heap, i.e. the least frequently used one, will be deleted.

The process of query processing with cache is composed of two parts: the training phase and testing phase. Depending on whether or not the cache can be modified in testing phase, the cache mechanism can be described as one of two types: static cache or dynamic cache. In a static cache, the training phase will fill the cache space by following a specific cache strategy. The content of the cache can not be modified during the testing phase. While in a dynamic cache, the cache can always be updated whatever the phase is.

6 Experiments

6.1 Experimental Setup

In our experiments, we use two real-world data sets as our text collection: GOV2 and ClueWeb09 (English pages in Category A). Collections GOV2 and ClueWeb09 contain approximately 25 million and 503 million documents, respectively. We analyze and format the web pages from these two raw text collections with our text preprocessing module. The stop words are filtered, and other words are stemmed with the Porter algorithm. The sizes of the two indexes are roughly 124 GB and 2.6 TB, respectively. The full index contains the posting lists file, position lists file, store fields file (containing all of the readable content of each page) and other auxiliary files. The value of option *SkipInterval* is set to 256, which means there are 256 docIDs in each posting list block (typically except the last block of a list).

In addition, we use the TREC Terabyte Track (100,000 queries from year 2006), and an AOL query log (100,000 queries) which was randomly extracted from 14.4 million queries log of 650,000 AOL users. We carry out all experiments on a PC server with a quad-core of a 3.40 GHz Intel(R) Core(TM) CPU and 32 GB of memory, running Centos 6.5. All components of the compression and cache framework are implemented in C++ and are compiled with g++ 4.8.2, with optimization flag -O2.

6.2 Encoding and Decoding

In this section, we compare the compression ratio and decompression speed for different compression methods. The compression ratio is measured as the average

Table 2. Space usage (bits per integer) and decoding speed (thousand integers per millisecond) for different encoding and decoding methods

Method	GOV2		ClueWeb09	
	Space	Speed	Space	Speed
Vbyte	6.32	587	8.50	341
GByte	6.72	793	9.24	557
GByte-SSE	6.80	819	9.41	572
GByte-AVX	6.91	802	9.75	562
S9	5.81	613	7.54	342
S9-SSE	5.81	633	7.54	360
S9-AVX	5.81	631	7.54	359
S16	5.67	599	6.87	289
S16-SSE	5.67	598	6.87	292
S16-AVX	5.67	600	6.87	292
NPFD	5.57	842	7.93	442
NPFD-SSE	5.54	901	7.94	521
NPFD-AVX	5.54	914	7.95	544

number of bits used per integer. The integers here refer to the elements of posting lists in NBLucene term frequency file, including docIDs and term frequencies. To compare the decompression speed, we execute the TREC and AOL query logs on the GOV2 and ClueWeb09 collections, respectively. During query processing, we count the number of integers actually decompressed and the time cost for decompression from code words to the original integers.

Table 2 shows that except from GByte methods, the other methods get better compression (GByte refers to GroupByte). GByte-SSE refers to varint-G8IU while GByte-AVX is a variant of varint-G8IU implemented using AVX instructions. S9, S16 and NPFD refers to the scalar Simple 9, Simple 16 and NewPForD. On the GOV2 collection, the best encoding method is NPFD-SSE and NPFD-AVX. They yield an enhancement of 12 % over the baseline VByte. On the ClueWeb09 collection, the best encoding method is S16 and its SIMD variants with an improvement is of 19 %. The highest decompression throughput of two datasets is achieved by NPFD-AVX and GByte-SSE. They are 56 % and 68 % faster than the baseline, respectively. We also find that the SIMD variants can get faster decompression speed than scalar version, and the improvement is up to 23 %. Although we only compare several methods in our experiment, the index compression module in NBLucene can be easily extended for other encoding methods as needed.

Table 3. Cache hit ratio and average query response time (milliseconds) for different results caches and cache sizes (MB)

Cache size	Cache type	GOV2		ClueWeb09	
		Ratio	Time	Ratio	Time
100	S-LRU	0.7 %	211	2.2 %	365
	S-LFU	0.5 %	211	1.9 %	370
	D-LRU	1.1 %	208	3.0 %	351
	D-LFU	1.0 %	210	2.4 %	358
200	S-LRU	0.8 %	209	2.4 %	355
	S-LFU	0.7 %	209	2.0 %	358
	D-LRU	1.8 %	206	3.1 %	344
	D-LFU	1.4 %	207	2.4 %	349
300	S-LRU	0.8 %	209	2.4 %	355
	S-LFU	0.7 %	209	2.0 %	357
	D-LRU	2.1 %	205	3.5 %	339
	D-LFU	1.5 %	207	2.6 %	345

6.3 Cache Framework

For the cache framework, we compare the cache hit ratio and average query response time with different results cache setups. The result of snippet cache is similar. For the ClueWeb09 collection, we extract the first 50 million documents to build a new index for running queries. We execute the TREC and AOL query logs on the whole GOV2 index and the new ClueWeb09 index, respectively. In each query set, we extract the first 50,000 queries for training and the other 50,000 queries for testing. We compare the static result cache and dynamic result cache with LRU and LFU strategies under different cache sizes.

In Table 3, we can see that the cache hit ratio improves and average query response time decreases with cache size increasing. The prefix used for the cache type name indicates the cache mechanism. For example, S-LRU indicates using a static cache with the LRU strategy while D-LFU indicating using a dynamic cache with the LFU strategy. We find that the dynamic caches outperform the static caches with the same configuration. Also, the caches with the LRU strategy have a higher hit ratio than the ones using the LFU strategy. Moreover, by designing the specific types of cache and replacement strategies, users can easily extend the result summarization module to implement their own cache experiments.

7 Conclusions and Future Work

In this paper, we design an efficient open source library for text searching. The most important contribution of our work is to provide a flexible architecture to

enable researchers to readily implement and modify search engine algorithms and strategies. For this purpose, we extend the CLucene by defining a series of typical experimental interfaces which involve the different stages during text searching. In addition to these interfaces, we also implement the parser for TREC web format and Web Archive format, the extensible mathematical encoding library and the cache framework with basic replacement strategies. In our experiments, we have compared the compression methods on two real-world datasets, GOV2 and ClueWeb09. We also compare the different cache strategies in our cache framework.

As future work, we plan to extend our core library to support more query operations, especially to support WAND queries and early termination in scored queries. Also, we aim to implement an architecture which could support query processing on GPU platforms.

References

1. Anh, V.N., Moffat, A.: Index compression using fixed binary codewords. In: Proceedings of 15th Australasian Database Conference, vol. 27, pp. 61–67 (2004)
2. Anh, V.N., Moffat, A.: Inverted index compression using word-aligned binary codes. Inf. Retrieval **8**, 151–166 (2005)
3. Ao, N., Zhang, F., Wu, D., Stones, D.S., Wang, G., Liu, X., Liu, J., Lin, S.: Efficient parallel lists intersection and index compression algorithms using graphics processing units. Proc. VLDB Endowment **4**(8), 470–481 (2011)
4. Bast, H., Celikik, M.: Efficient index-based snippet generation. ACM Trans. Inf. Syst. **32**(2), 6 (2014)
5. Büttcher, S., Clarke, C.L.A., Cormack, G.V.: Information Retrieval: Implementing and Evaluating Search Engines. MIT Press, Cambridge (2010)
6. Cutting, D., Pedersen, J.: Optimization for dynamic inverted index maintenance. In: Proceedings of 13th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 405–411 (1989)
7. Dean, J.: Challenges in building large-scale information retrieval systems. In: Proceedings of 2nd ACM International Conference on Web Search and Web Data Mining, WSDM 2009, p. 1. ACM (2009)
8. Ding, S., He, J., Yan, H., Suel, T.: Using graphics processors for high performance IR query processing. In: Proceedings of 18th International Conference on World Wide Web, pp. 421–430 (2009)
9. Lemire, D., Boytsov, L.: Decoding billions of integers per second through vectorization. Softw. Pract. Exp. **45**, 1–29 (2015)
10. Lemire, D., Boytsov, L., Kurz, N.: SIMD compression and the intersection of sorted integers. CoRR, abs/1401.6399 (2014)
11. Middleton, C., Baeza-Yates, R.: A comparison of open source search engines. Technical report, Department of Technologies, Universitat Pompeu Fabra (2007)
12. Robertson, S.E., Jones, K.S.: Relevance weighting of search terms. J. Am. Soc. Inform. Sci. Technol. **27**(3), 129–146 (1976)
13. Schlegel, B., Willhalm, T., Lehner, W.: Fast sorted-set intersection using SIMD instructions. In: International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures, pp. 1–8 (2011)

14. Stepanov, A.A., Gangolli, A.R., Rose, D.E., Ernst, R.J., Oberoi, P.S.: SIMD-based decoding of posting lists. In: Proceedings of 20th ACM International Conference on Information and Knowledge Management, pp. 317–326 (2011)
15. Strohman, T., Metzler, D., Turtle, H., Croft, W.B.: Indri: a language model-based search engine for complex queries. In: Proceedings of International Conference on Intelligent Analysis, vol. 2, pp. 2–6 (2005)
16. Turpin, A., Tsegay, Y., Hawking, D., Williams, H.E.: Fast generation of result snippets in web search. In: Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 127–134 (2007)
17. Varadarajan, R., Hristidis, V.: A system for query-specific document summarization. In: Proceedings of 15th ACM International Conference on Information and Knowledge Management, pp. 622–631 (2006)
18. Willhalm, T., Popovici, N., Boshmaf, Y., Plattner, H., Zeier, A., Schaffner, J.: SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. Proc. VLDB Endowment **2**, 385–394 (2009)
19. Yan, H., Ding, S., Suel, T.: Inverted index compression and query processing with optimized document ordering. In: Proceedings of 18th International Conference on World Wide Web, pp. 401–410 (2009)
20. Zhang, J., Long, X., Suel, T.: Performance of compressed inverted list caching in search engines. In: Proceedings of 17th International Conference on World Wide Web, pp. 387–396 (2008)
21. Zhang, X., Zhao, W.X., Shan, D., Yan, H.: Group-scheme: SIMD-based compression algorithms for web text data. In: Proceedings of 2013 IEEE International Conference on Big Data, pp. 525–530 (2013)
22. Zobel, J., Williams, H., Scholer, F., Yiannis, J., Hein, S.: The Zettair Search Engine. Search Engine Group, RMIT University, Melbourne (2004)