PISA: Performant Indexes and Search for Academia

Antonio Mallia
antonio.mallia@nyu.edu
New York University
New York, US

Michał Siedlaczek
michal.siedlaczek@nyu.edu
New York University
New York, US

Joel Mackenzie
joel.mackenzie@rmit.edu.au
RMIT University
Melbourne, Australia

Torsten Suel
torsten.suel@nyu.edu
New York University
New York, US

ABSTRACT

Performant Indexes and Search for Academia (PISA) is an experimental search engine that focuses on *efficient* implementations of state-of-the-art representations and algorithms for text retrieval. In this work, we outline our effort in creating a replicable search run from PISA for the 2019 *Open Source Information Retrieval Replicability Challenge*, which encourages the information retrieval community to produce replicable systems through the use of a containerized, Docker-based infrastructure. We also discuss the origins, current functionality, and future direction and challenges for the PISA system.

KEYWORDS

Search Engine, Open-Source System, Replicability

1 INTRODUCTION

Reproducibility, replicability, and generalizability have become increasingly important within the Information Retrieval (IR) community. For example, *weak baselines* [3, 18] are often used as comparison points against newly proposed systems, resulting in what often appear to be large improvements. One possible reason that weak baselines are used is that they are usually simple and well established, making it easy to reproduce or replicate them. Indeed, replicating experimental runs is not a trivial task; minor changes in software, datasets, and even hardware can result in significant changes to experimental runs [10]. To this end, the 2019 Open Source Information Retrieval Replicability Challenge (OSIRRC) brings together academic groups with the aim of defining a reusable framework for easily running IR experiments with a particular focus on *replicability*, where a *different* team (to those who proposed the system) uses the *original* experimental artifacts to *replicate* the given result. In an attempt to improve replicability, the OSIRRC workshop proposes to package and deploy various IR systems within a *Docker container*,[1] which enables low-effort replication by reducing many experimental confounders.

The goal of this paper is to give an overview of the PISA system and to outline the process of building replicable runs from PISA with Docker. We outline the particulars of our submitted runs, and discuss where PISA is suited for use in IR experimentation. The remainder of this paper is structured as follows. In Section 2 we describe some of the core functionality that makes PISA the state-of-the-art for efficient search. Section 3 outlines the particular runs that were deployed for the workshop, and shows some reference experiments across a variety of collections. In Section 4, we briefly outline the future of the PISA system. Finally, we conclude this work in Section 5.

[1]https://www.docker.com/

2 THE PISA ENGINE

PISA[2] is an open source library implementing text indexing and search, primarily targeted for use in an academic setting. PISA implements a range of state-of-the-art indexing and search algorithms, making it useful for researchers to easily experiment with new technologies, especially those concerned with efficiency. Nevertheless, we strive for much more than just an efficient implementation. With clean and extensible design and API, PISA provides a general framework that can be employed for miscellaneous research tasks, such as parsing, ranking, sharding, index compression, document reordering and query processing, to name a few.

PISA started off as a fork repository of the ds2i library[3] by Ottaviano et al., which was used for prior research on efficient index representations [26, 27]. Since then, PISA has gone through substantial changes, and now considerably extends the original library. PISA is still being actively developed, integrating new features and improving its design and implementation at regular intervals.

2.1 Design Overview

PISA is designed to be efficient, extensible, and easy to use. We now briefly outline some of the design aspects of PISA.

Modern Implementation. The PISA engine itself is built using C++17, making use of many new features in the C++ standard. This allows us to ensure the implementations are both efficient and understandable, as some of the newest language features can make for cleaner code and APIs. We aim to adhere to best design practices, such as RAII (Resource Acquisition Is Initialization), C++ Core Guidelines[4] (aided by Guidelines Support Library[5]), and strongly-typed aliases, all of which result in safer and cleaner code without sacrificing runtime performance.

[2]https://https://github.com/pisa-engine/pisa
[3]https://github.com/ot/ds2i
[4]https://github.com/isocpp/CppCoreGuidelines
[5]https://github.com/Microsoft/GSL

Performance. One of the biggest advantages of C++ is its performance. Control over data memory layout allows us to implement and store efficient data structures with little to no runtime overhead. Furthermore, we make use of low level optimizations, such as CPU intrinsics, branch prediction hinting, and SIMD instructions, which are especially important for efficiently encoding and decoding postings lists. Memory mapped files provide fast and easy access to data structures persisted on disk. We also avoid unnecessary indirection of runtime polymorphism in performance-sensitive code in favor of the static polymorphism of C++ templates and metaprogramming. Our performance is also much more predictable than when using languages with garbage collection. Finally, we suffer no runtime overhead as is the case with VM-based or interpreted languages.

Extensibility. Another important design aspect of PISA promotes extensibility. For example, interfaces are exposed which allow for new components to be plugged in easily, such as different parsers, stemmers, and compression codecs. This is achieved through heavy use of generic programming, similar to that provided by the C++ Standard Template Library. For example, an encoding schema is as much a parameter of an index as a custom allocator is a parameter of `std::vector`. By decoupling different parts of the framework, we provide an easy way of extending the library both in future iterations of the project, as well as by users of the library.

2.2 Feature Overview

In this section, we take a short tour of several important features of our system. We briefly discuss the indexing pipeline, document reordering, scoring, and implemented retrieval methods.

Parsing Collection. The objective of parsing is to represent a given collection as a *forward index*, where each term is assigned a unique numerical ID, and each document consists of a list of such identifiers. This is a non-trivial task that involves several steps that can be critical to retrieval performance.

First, the document content must be accessed by parsing a certain data exchange format, such as *WARC*, *JSON*, or *XML*. The document itself is typically represented by *HTML*, *XML*, or a custom annotated format, which must be parsed to retrieve the underlying text. The text must be then tokenized, and the resulting words are typically stemmed before indexing.

PISA supports a selection of file and content parsers. The parsing tool allows input formats of many standard IR collections, such as ClueWeb09[6], ClueWeb12[7], GOV2[8], Robust04[9], Washington Post[10], and New York Times.[11] We also provide an HTML content parser, and the Porter2 [31] stemming algorithm for English language. As discussed in Section 2.1, PISA is designed to allow new components, such as parsers or stemmers, to be plugged-in with low implementation overhead.

As part of a forward index, we also encode a term *lexicon*. This is simply a mapping between strings and numerical IDs. We represent it as a *payload vector*. The structure is divided into two memory areas: the first one is an array of integers representing payload offsets, while the other stores the payloads (strings). This representation allows us to quickly retrieve a word at a given position—which determines its ID—directly from disk using memory mapping. We achieve string lookups by assigning term IDs in lexicographical order and performing binary search. Note that reassigning term IDs requires little overhead, as it is applied directly when a number of small index batches are merged together. This design decision enables us to provide a set of command line tools to quickly access index data without unnecessary index loading overhead. Document titles (such as TREC IDs) are stored using the same structure but without sorting them first, as the order of the documents is established via an index reordering stage as described below.

The entire parsing process is performed in parallel when executed on a multi-core architecture. The forward index can be created under tight memory constraints by dividing the corpus and processing it in batches, and then merging the resulting forward indexes at the end. Currently, PISA only supports merging forward indexes together prior to generating the canonical inverted index. However, future work will aim to allow index updates through efficient inverted index merging operations.

Indexing. Once the parsing phase is complete, the forward index containing a collection can be used to build an inverted index in a process called *inverting*. The product of this phase is an inverted index in the *canonical format*. This representation is very similar to the forward index, but in reverse: it is a collection of terms, each a containing a list of document IDs. The canonical representation is stored in an uncompressed and universally readable format, which simply uses *binary sequences* to represent lists. There are a few advantages of storing the canonical representation. Firstly, it allows various transformations, such as document reordering or index pruning, to be performed on the index before storing it in its final compressed form. Secondly, it allows for different final representations to be built rapidly, such as indexes that use different compression algorithms. Thirdly, it allows the PISA system to be used to create an inverted index *without* requiring the PISA system to be used beyond this point, making it easy for experimenters to import the canonical index into their own IR system.

Document Reordering. Document reordering corresponds to reassigning the document identifiers within the inverted index [4]. It generally aims to minimize the cost of representing the inverted index with respect to storage consumption. However, reordering can also be used to minimize other cost functions, such as query processing speed [41]. Interestingly, optimizing for space consumption has been empirically shown to speed up query processing [14, 15, 24], making document reordering an attractive step during indexing. In theory, index reordering can take place either on an existing inverted index, or before the inverted index is constructed. PISA opts to use the canonical index as both input and output for the document reordering step, as this allows a generic reordering scheme to be used which can be easily extended to various reordering techniques, and allows the reordering functionality to be used without requiring further use of PISA.

Many approaches for document reordering exist, including random ordering, reordering by URL [33], MinHash ordering [6, 9],

[6]https://lemurproject.org/clueweb09/
[7]https://lemurproject.org/clueweb12/
[8]http://ir.dcs.gla.ac.uk/test_collections/gov2-summary.htm
[9]https://trec.nist.gov/data/robust/04.guidelines.html
[10]https://trec.nist.gov/data/wapost/
[11]https://catalog.ldc.upenn.edu/LDC2008T19

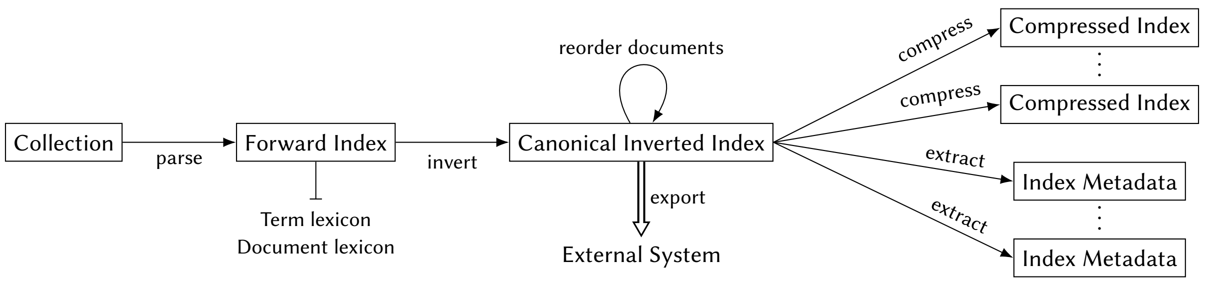

Figure 1: Index building pipeline in PISA. A collection is first parsed and encoded in a forward index. Subsequently, it is *inverted* and stored in the canonical inverted index format. This can be used to efficiently reorder documents. Eventually, a compressed representation is produced, which will be used at query time. Additional data might be extracted, depending on the algorithms used. The simplicity of the inverted index format (uncompressed) makes it easy to convert it to any given format used by another framework.

and recursive graph bisection [13]. PISA supports efficient index reordering for all of the above techniques [21].

Index Compression. Given the extremely large collections indexed by current search engines, even a single node of a large search cluster typically contains many billions of integers in its index structure. In particular, compression of posting lists is of utmost importance, as they account for much of the data size and access costs. Compressing an inverted index introduces a twofold advantage over a non-compressed representation: it reduces the size of the index, and it allows us to better exploit the memory hierarchy, which consequently speeds up query processing.

Compression allows the space requirements of indexes to be substantially decreased without loss of information. The simple and extensible design of PISA allows for new compression approaches to be plugged in easily. As such, a range of state-of-the-art compression schemes are currently supported, including *variable byte* encoders (VarIntGB [12], VarInt-G8IU [34], MaskedVByte [30], and StreamVByte [17]), *word-aligned* encoders (Simple8b [2], Simple16 [43], QMX [35, 37], and SIMD-BP128 [16]), *monotonic* encoders (Interpolative [25], EF [40], and PEF [27]), and *frame-of-reference* encoders (Opt-PFD [42]).

Oftentimes, the choice of encoding depends on both the time and space constraints, as compression schemes usually trade off space efficiency for either encoding or decoding performance, or both. We refer the reader to [24] for more details.

Scoring. Currently, BM25 [32] is used as the weighting model for ranked retrieval. BM25 is a simple yet effective ranker for processing bag-of-words queries, and promotes effective dynamic pruning [28]. Given a document d and a query q, we use the following formulation of BM25:

$$\text{BM25}(d,q) = \sum_{t \in q} \text{IDF}(t) \cdot \text{TF}(d,t), \tag{1}$$

$$\text{IDF}(t) = \log\left(\frac{N - f_t + 0.5}{f_t + 0.5}\right), \tag{2}$$

$$\text{TF}(d,t) = \frac{f_{d,t} \cdot (k_1 + 1)}{f_{d,t} + k_1 \cdot (1 - b + b \cdot l_d / l_{\text{avg}})}, \tag{3}$$

where N is the number of documents in the collection, f_t is the document frequency of term t, $f_{d,t}$ is the frequency of t in d, l_d is the length of document d, and l_{avg} is the average document length. We set parameters $k_1 = 0.9$ and $b = 0.4$, as described by Trotman et al. [36]. For a more exhaustive examination of BM25 variants, we refer the reader to the work by Trotman et al. [38].

Search. Because PISA supports *document-ordered* index organization, both Boolean and scored conjunctions or disjunctions can be evaluated, exploiting either a Document-at-a-Time or a Term-at-a-Time index traversal strategy.

Furthermore, PISA supports a range of state-of-the-art *dynamic pruning* algorithms such as MaxScore [39] and WAND [5], and their *Block-Max* counterparts, Block-Max MaxScore (BMM) [7] and Block-Max WAND (BMW) [14].

In order to facilitate these dynamic pruning algorithms, an auxiliary *index metadata* structure must be built, which stores the required upper-bound score information to enable efficient dynamic pruning. It can be built per postings list (for algorithms like WAND and MaxScore), or for each fixed-sized block (for the Block-Max variants). In addition, variable-sized blocks can be built (in lieu of fixed-sized blocks) to support the *variable-block* family of Block-Max algorithms listed above, such as Variable Block-Max WAND (VBMW) [22, 23]. Ranked conjunctions are also supported using the Ranked AND or (Variable) Block-Max Ranked AND (BMA) [14] algorithms.

Indeed, the logical blocks stored in the index metadata are decoupled from the compressed blocks inside the inverted index. The advantage of storing the metadata independently from the inverted index is that the metadata depends only on the canonical index, needs to be computed only *once*, and does not change if the underlying compression codec is changed.

PISA provides two ways to experiment with query retrieval. The first one performs end-to-end search for a given list of queries, and prints out the results in the TREC format. It can also be used to evaluate query speed, as was done for this workshop. Additionally, we provide a more granular approach, which focuses on comparing different retrieval methods directly. Here, we only report the time to fetch posting lists and perform search, excluding lexicon lookups and printing results to the standard output or a file. We encourage

Robust04	Core17	Core18	Gov2	ClueWeb09	ClueWeb12
16.5	15	13.5	17	22.5	25.5

Table 1: Values of λ for the given collections using the Gumbo parser, Porter2 stemmer, and reordering with recursive graph bisection. These values will yield an average block size of 40 ± 0.5 for the variable block metadata.

the interested reader to refer to PISA's documentation for more details about running experiments.[12]

3 REPRODUCIBLE EXPERIMENTATION

In the spirit of OSIRRC, we utilize the software and metrics made available by the organizers, including the `jig`[13] and the `trec_eval`[14] tool. In addition, we have decided to provide some further information and reference experiments that we consider important.

3.1 Default Runs

Given the many possibilities for the various components of the PISA system, we now outline the *default* system configuration for the OSIRRC workshop. Further information can be found in the documentation of our OSIRRC system.[15] Note that the block size for the variable-block indexes depends on a parameter λ [22]. In order to get the desired average block size for the variable blocks, the value of λ was searched for offline, and differs for each collection. For convenience, we tabulate the values of λ in Table 1. Note that such values of λ only apply when using the same parsing, stemmer, and reordering as listed below.

- **Parsing:** Gumbo[16] with Porter2 stemming; no stopwords removed. We discard the content of any document with over 1,000 HTML parsing errors.
- **Reordering:** Recursive graph bisection. We optimize the objective function using the posting lists of lengths at least 4,096.
- **Compression:** SIMD-BP128 with a posting list block size of 128.
- **Scoring:** BM25 with $k_1 = 0.9$ and $b = 0.4$
- **Index Metadata:** Variable blocks with a mean block size of 40 ± 0.5.
- **Index Traversal:** Variable BlockMax WAND.

3.2 Experimental Setup

Now, we briefly outline the experimental setup and the resources used for our experimentation.

Datasets. We performed our experiments on the following text collections:

- Robust04 consists of newswire articles from a variety of sources from the late 1980's through to the mid 1990's.

[12]https://pisa.readthedocs.io/en/latest/
[13]https://github.com/osirrc/jig/
[14]https://github.com/usnistgov/trec_eval
[15]https://github.com/osirrc/pisa-docker
[16]https://github.com/google/gumbo-parser

	Documents	Terms	Postings
Robust04	528,155	587,561	107,481,358
Core17	1,855,660	1,048,294	448,998,765
Core18	595,037	621,941	191,042,917
Gov2	25,205,178	32,407,061	5,264,576,636
ClueWeb09	50,220,110	87,896,391	14,996,638,171
ClueWeb12	52,343,021	133,248,235	14,173,094,439

Table 2: Quantitative properties of our indexes.

Collection	Track	Topics	# Topics
Robust04	Robust '04	301–450, 601–700	250
Core17	Common Core '17	301–450, 601–700†	250
Core18	Common Core '18	321–825‡	50
Gov2	Terabyte '04-'06	701–850	150
ClueWeb09	Web '09-'12	51–200	150
ClueWeb12	Web '13-'14	201–300	100

Table 3: Query topics used in the experiments. Note that the Core17 topics are the same as the Robust04 topics, but some were modified to reflect temporal changes†, and Core18 used 25 topics from Core17 and 25 new topics.‡

- Core17 corresponds to the New York Times news collection, which contains news articles between 1987 and 2007.
- Core18 is the TREC Washington Post Corpus, which consists of news articles and blog posts from January 2012 through August 2017.
- Gov2 is the TREC 2004 Terabyte Track test collection consisting of around 25 million `.gov` sites crawled in early 2004; the documents are truncated to 256 kB.
- ClueWeb09 is the ClueWeb 2009 Category B collection consisting of around 50 million English web pages crawled between January and February, 2009.
- ClueWeb12 is the ClueWeb 2012 Category B-13 collection, which contains around 52 million English web pages crawled between February and May, 2012.

Some quantitative properties of these collections are summarized in Table 2. The first three are relatively small, and contain newswire data. The remaining corpora are significantly larger, containing samples of the Web. Thus, the latter two should be more indicative of any differences in query efficiency. In fact, each of these can be thought of as representing a single shard in a large distributed search system.

Test queries. Each given collection contains a set of test queries from various TREC tracks which we use to validate the effectiveness of our system. These queries are described in Table 3.

Testing details. All experiments were conducted on a machine with two Intel Xeon E5-2667 v2 CPUs, with a total of 32 cores clocked at 3.30 GHz with 256 GiB RAM running Linux 4.15.0. Furthermore, the experiments presented here are deployed *within* the

Docker framework. Although we believe that this may cause a slight reduction in the efficiency of the presented algorithms, we preserve this setup in the spirit of the workshop and comparability. We leave further investigation of potential overhead of Docker containers as future work.

A note on ClueWeb12. In preliminary experiments, we found that the memory consumption for reordering the ClueWeb12 index was high, which slowed down the indexing process considerably. Thus, we opted to skip reordering the ClueWeb12 collection in the following experiments, and our results are reported on an index that uses the default (crawl) order. Since index order impacts the value of λ, we use $\lambda = 26$, which results in variable block metadata with a mean block size in the desired range of 40 ± 0.5. Note that this value differs from the one reported in Table 1, which is correct if reordering is applied based on Recursive Graph Bisection (see Section 3.1).

3.3 Results and Discussion

We now present our reference experiments, which involve end-to-end processing of each given collection.

Indexing and Compression. The HTML content of each document was extracted with the Gumbo parser. We then extracted three kinds of tokens: alphanumeric strings, acronyms, and possessives, which were then stemmed using the Porter2 algorithm. We reordered documents using the recursive graph bisection algorithm which is known to improve both compression and query performance [13, 21, 24]. Then we compressed the index with SIMD-BP128 encoding, which has been proven to exhibit one of the best space-speed trade offs [24].

Table 4 summarizes indexing times broken down into individual phases, while Table 5 shows compressed inverted index sizes as well as average numbers of bits used to encode document gaps and frequencies. The entire building process was executed with 32 cores available; however, at the time of writing, only some parts of the pipeline support parallel execution. We also note that the index reordering step is usually the most expensive step in our indexing pipeline. If a fast indexing time is of high importance, this step can be omitted, as we did for ClueWeb12. Alternatively, less expensive reordering operations can be used. However, skipping the index reordering stage (or using a less effective reordering technique) will result in a larger inverted index and less efficient query-time performance.

System Effectiveness. Next, we outline the *effectiveness* of the PISA system. In particular, we are processing *rank-safe, disjunctive, top-k* queries to depth $k = 1{,}000$. Since processing is rank-safe, all of the disjunctive index traversal algorithms result in the same top-k set. Table 6 reports the effectiveness for *Mean Average Precision* (MAP), *Precision* at rank 30 (P@30), and *Normalized Discounted Cumulative Gain* at rank 20 (NDCG@20).

Query Efficiency. To measure the efficiency of query processing, we measure how long it takes to process the *entire* query log for each collection. We use 32 threads to concurrently retrieve the top-k documents for all queries using either the MaxScore or the VBMW algorithm, with a single thread processing a single query at a time. MaxScore has been shown to outperform other algorithms for large values of k on the Gov2 and ClueWeb09 collections [24]. Table 7 shows the results. While MaxScore usually outperforms VBMW, we did not optimize the block size of the index metadata, so comparisons should be made with caution. Indeed, VBMW is likely to outperform MaxScore with optimized blocks and small values of k. For a more detailed analysis of per-query latency within PISA, we refer the interested reader to the recent work by Mallia et al. [24].

3.4 Discussion

PISA is built for performance. We are able to rapidly process each query set thanks to efficient document retrieval algorithms and extremely fast compression. On the other hand, as we have shown, SIMD-BP128 encoding also exhibits a reasonable compression ratio, which allows us to store the index in main memory. We encourage the reader to study the work by Mallia et al. [24] for more information about query efficiency under different retrieval and compression methods.

At the present moment, our query retrieval is tailored towards fast candidate selection, as we lack any complex ranking functionality, such as a learning-to-rank document reranking stage. However, the effectiveness we obtain using BM25 is consistent with other results found in the literature [19].

Furthermore, we provide a generic index building pipeline, which can be easily customized to one's needs. We unload most of the computationally intensive operations onto the initial stages of indexing to speed up experiments with many configurations; in particular, to deliver additional indexes with different integer encodings quickly and easily.

As per the workshop rules, we deliver a Docker image, which reproduces the presented results. Note that the initial version of the image was derived from an image with a precompiled distribution of PISA. However, we quickly discovered this solution was not portable. The source of our issues was compiling the code with AVX2 support. Once compiled, the binaries could not be executed on a machine not supporting AVX2. One solution could be to cross-compile and provide different versions of the image. However, we chose to simply distribute the source code to be compiled at the initial stage of an experimental run.

4 FUTURE PLANS

Despite its clear strengths, PISA is still a relatively young project, aspiring to become a more widely used tool for IR experimentation. We recognize that many relevant features can be still developed to further enrich the framework. We have every intention of pursuing these in the nearest future.

An obvious direction is to continue our work on query performance. For instance, we intend to support precomputing quantized partial scores in order to further improve candidate selection performance [11]. We are also considering implementing other traversal techniques, including known approaches, such as Score-at-a-Time methods [1, 20], as well as novel techniques.

The next step would be to implement more complex document rankings based on learning-to-rank. Many of the data structures required for feature extraction are indeed already in place. We

	Parse	Invert	Reorder	Compress	Metadata	**Total**
Robust04	0:06:22	0:00:08	0:01:21	0:00:07	0:00:44	0:08:42
Core17	0:11:41	0:00:42	0:07:18	0:00:14	0:02:59	0:22:54
Core18	0:10:42	0:00:14	0:02:11	0:00:07	0:01:11	0:14:25
Gov2	1:37:52	1:00:12	2:28:04	0:06:42	0:37:04	5:49:52
ClueWeb09	4:08:08	3:11:50	10:28:30	0:32:42	2:01:01	20:22:12
ClueWeb12	5:09:58	3:27:51	—	0:34:55	2:11:46	11:24:30

Table 4: Indexing times broken down into five phases: parsing, inverting, reordering, compression, and index metadata construction. Times are reported in the following format: *hours:minutes:seconds*.

	Index size (MiB)	Docs (bpi)	Freqs (bpi)
Robust04	136.88	7.48	3.21
Core17	545.90	7.07	3.13
Core18	238.33	7.24	3.22
Gov2	5,410.89	5.59	3.03
ClueWeb09	20,715.29	7.96	3.63
ClueWeb12	23,206.16	9.22	4.52

Table 5: Total index size and average number of bits per integer while encoding documents and frequencies within posting lists.

	$k = 10$		$k = 1{,}000$	
Collection	MaxScore	VBMW	MaxScore	VBMW
Robust04	7	7	21	26
Core17	10	8	16	15
Core18	5	4	12	14
Gov2	115	99	215	200
ClueWeb09	225	138	285	424
ClueWeb12	220	415	248	842

Table 7: Time taken to process the entire query log for each collection. Time is reported in milliseconds.

	Topics	MAP	P@30	NDCG@20
Robust04	All	0.2534	0.3120	0.4221
Core17	All	0.2078	0.4260	0.3898
Core18	All	0.2384	0.3500	0.3927
Gov2	701-750	0.2638	0.4776	0.4070
	751-800	0.3305	0.5487	0.5073
	801-850	0.2950	0.4680	0.4925
ClueWeb09	51-100	0.1009	0.2521	0.1509
	101-150	0.1093	0.2507	0.2177
	151-200	0.1054	0.2100	0.1311
ClueWeb12	201-250	0.0449	0.1940	0.1529
	251-300	0.0217	0.1240	0.1484

Table 6: The effectiveness of the submitted run for each respective corpus.

would also like to enhance our query retrieval pipeline with ranking cascades that are capable of applying learned models [8].

Other planned features include query expansion, content extraction (template detection, boilerplate removal), sharding, and distributed indexes. Work on some of these has in fact already started.

5 CONCLUSION

PISA is a relative newcomer on the scene of open source IR software, yet it has already proven its many benefits, including a flexible design which is specifically tailored for use in research. Indeed, PISA has been successfully used in several recent research papers [21, 23, 24, 29].

One of the indisputable advantages of PISA is its extremely fast query execution, achieved by careful optimization and the zero-cost abstractions of C++. Furthermore, it supports a multitude of state-of-the-art compression and query processing techniques that can be used interchangeably.

Although there are still several shortcomings, these are mostly due to the project's young age, and we hope to address these very soon. Furthermore, we plan to continue enhancing the system with novel solutions. Indeed, a good amount of time has been spent on PISA to provide a high quality experimental IR framework, not only in terms of performance, but also from a software engineering point of view. We use modern technologies and libraries, continuous integration, and test suites to ensure the quality of our code, and the correctness of our implementations.

We encourage any interested researchers to get involved with the PISA project.

ACKNOWLEDGEMENTS

This research was supported by NSF Grant IIS-1718680, a grant from Amazon and the Australian Research Training Program Scholarship.

REFERENCES

[1] V. N. Anh, O. de Kretser, and A. Moffat. 2001. Vector-space ranking with effective early termination.. In *Proc. SIGIR*. 35–42.

[2] V. N. Anh and A. Moffat. 2010. Index compression using 64-bit words. *Soft. Prac. & Exp.* 40, 2 (2010), 131–147.

[3] T. G. Armstrong, A. Moffat, W. Webber, and J. Zobel. 2009. Improvements that don't add up: Ad-hoc Retrieval Results since 1998. In *Proc. CIKM*. 601–610.

[4] D. Blandford and G. Blelloch. 2002. Index Compression Through Document Reordering. In *Proc. DCC*. 342–352.

[5] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. 2003. Efficient Query Evaluation Using a Two-level Retrieval Process. In *Proc. CIKM*. 426–434.

[6] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. 2000. Min-Wise Independent Permutations. *J. Comput. Syst. Sci.* 60, 3 (2000), 630–659.

[7] K. Chakrabarti, S. Chaudhuri, and V. Ganti. 2011. Interval-based pruning for top-k processing over compressed lists. In *Proc. ICDE*. 709–720.

[8] R-C. Chen, L. Gallagher, R. Blanco, and J. S. Culpepper. 2017. Efficient Cost-Aware Cascade Ranking in Multi-Stage Retrieval. In *Proc. SIGIR*. 445–454.

[9] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. 2009. On Compressing Social Networks. In *Proc. SIGKDD*. 219–228.

[10] M. Crane. 2018. Questionable Answers in Question Answering Research: Reproducibility and Variability of Published Results. *Trans. ACL* 6 (2018), 241–252.

[11] M. Crane, A. Trotman, and R. O'Keefe. 2013. Maintaining discriminatory power in quantized indexes. In *Proc. CIKM*. 1221–1224.

[12] J. Dean. 2009. Challenges in Building Large-scale Information Retrieval Systems: Invited Talk. In *Proc. WSDM*. 1–1.

[13] L. Dhulipala, I. Kabiljo, B. Karrer, G. Ottaviano, S. Pupyrev, and A. Shalita. 2016. Compressing Graphs and Indexes with Recursive Graph Bisection. In *Proc. KDD*. 1535–1544.

[14] S. Ding and T. Suel. 2011. Faster top-k document retrieval using block-max indexes. In *Proc. SIGIR*. 993–1002.

[15] D. Hawking and T. Jones. 2012. Reordering an Index to Speed Query Processing Without Loss of Effectiveness. In *Proc. ADCS*. 17–24.

[16] D. Lemire and L. Boytsov. 2015. Decoding billions of integers per second through vectorization. *Soft. Prac. & Exp.* 45, 1 (2015), 1–29.

[17] D. Lemire, N. Kurz, and C. Rupp. 2018. Stream VByte: Faster byte-oriented integer compression. *Inf. Proc. Letters* 130 (2018), 1–6.

[18] J. Lin. 2019. The Neural Hype and Comparisons Against Weak Baselines. *SIGIR Forum* 52, 2 (2019), 40–51.

[19] J. Lin, M. Crane, A. Trotman, J. Callan, I. Chattopadhyaya, J. Foley, G. Ingersoll, C. Macdonald, and S. Vigna. 2016. Toward Reproducible Baselines: The Open-Source IR Reproducibility Challenge. In *Proc. ECIR*. 408–420.

[20] J. Lin and A. Trotman. 2015. Anytime Ranking for Impact-Ordered Indexes. In *Proc. ICTIR*. 301–304.

[21] J. Mackenzie, A. Mallia, M. Petri, J. S. Culpepper, and T. Suel. 2019. Compressing Inverted Indexes with Recursive Graph Bisection: A Reproducibility Study. In *Proc. ECIR*. 339–352.

[22] A. Mallia, G. Ottaviano, E. Porciani, N. Tonellotto, and R. Venturini. 2017. Faster BlockMax WAND with Variable-sized Blocks. In *Proc. SIGIR*. 625–634.

[23] A. Mallia and E. Porciani. 2019. Faster BlockMax WAND with longer skipping. In *Proc. ECIR*. 771–778.

[24] A. Mallia, M. Siedlaczek, and T. Suel. 2019. An Experimental Study of Index Compression and DAAT Query Processing Methods. In *Proc. ECIR*. 353–368.

[25] A. Moffat and L. Stuiver. 2000. Binary Interpolative Coding for Effective Index Compression. *Inf. Retr.* 3, 1 (2000), 25–47.

[26] G. Ottaviano, N. Tonellotto, and R. Venturini. 2015. Optimal Space-time Tradeoffs for Inverted Indexes. In *Proc. WSDM*. 47–56.

[27] G. Ottaviano and R. Venturini. 2014. Partitioned Elias-Fano indexes. In *Proc. SIGIR*. 273–282.

[28] M. Petri, J. S. Culpepper, and A. Moffat. 2013. Exploring the Magic of WAND. In *Proc. Aust. Doc. Comp. Symp.* 58–65.

[29] M. Petri, A. Moffat, J. Mackenzie, J. S. Culpepper, and D. Beck. 2019. Accelerated Query Processing Via Similarity Score Prediction. In *Proc. SIGIR*. To Appear.

[30] J. Plaisance, N. Kurz, and D. Lemire. 2015. Vectorized VByte Decoding. In *Int. Symp. Web Alg.*

[31] M. F. Porter. 1997. Readings in IR. Chapter An Algorithm for Suffix Stripping, 313–316.

[32] S. E. Robertson, S. Walker, S. Jones, M. Hancock-Beaulieu, and M. Gatford. 1994. Okapi at TREC-3. In *Proc. TREC*.

[33] F. Silvestri. 2007. Sorting out the Document Identifier Assignment Problem. In *Proc. ECIR*. 101–112.

[34] A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. Oberoi. 2011. SIMD-based Decoding of Posting Lists. In *Proc. CIKM*. 317–326.

[35] A. Trotman. 2014. Compression, SIMD, and Postings Lists. In *Proc. Aust. Doc. Comp. Symp.* 50.50–50.57.

[36] A. Trotman, X-F. Jia, and M. Crane. 2012. Towards an efficient and effective search engine. In *Wkshp. Open Source IR*. 40–47.

[37] A. Trotman and J. Lin. 2016. In Vacuo and In Situ Evaluation of SIMD Codecs. In *Proc. Aust. Doc. Comp. Symp.* 1–8.

[38] A. Trotman, A. Puurula, and B. Burgess. 2014. Improvements to BM25 and Language Models Examined. In *Proc. Aust. Doc. Comp. Symp.* 58–65.

[39] H. R. Turtle and J. Flood. 1995. Query Evaluation: Strategies and Optimizations. *Inf. Proc. & Man.* 31, 6 (1995), 831–850.

[40] S. Vigna. 2013. Quasi-succinct indices. In *Proc. WSDM*. 83–92.

[41] Q. Wang and T. Suel. 2019. Document Reordering for Faster Intersection. *Proc. VLDB* 12, 5 (2019), 475–487.

[42] H. Yan, S. Ding, and T. Suel. 2009. Inverted index compression and query processing with optimized document ordering. In *Proc. WWW*. 401–410.

[43] J. Zhang, X. Long, and T. Suel. 2008. Performance of Compressed Inverted List Caching in Search Engines. In *Proc. WWW*. 387–396.