JASSjr: The Minimalistic BM25 Search Engine for Teaching and Learning Information Retrieval

Andrew Trotman
University of Otago
Dunedin, New Zealand

Kat Lilly
University of Otago
Dunedin, New Zealand

ABSTRACT

We present JASSjr, a minimalistic `trec_eval` compatible BM25-ranking search engine that can index small TREC data sets such as the Wall Street Journal collection. We do this for several reasons. First, to demonstrate how a term-at-a-time (TaaT) search engine works. Second, to demonstrate that a straightforward and competitive search engine with indexer can be written in under 600 lines of documented code. Third, as a way of providing a simple code-base for teaching Information Retrieval. We present two index-compatible versions (one in C/C++, the other in Java) that compile and run on MacOS, Linux, and Windows.

Our code is released under the 2-clause BSD licence, and we provide several suggestions for extensions which might be used as exercises in an Information Retrieval course.

CCS CONCEPTS

• **Information systems → Search engine architectures and scalability**.

KEYWORDS

Inverted index; TaaT-search; BM25

ACM Reference Format:
Andrew Trotman and Kat Lilly. 2020. JASSjr: The Minimalistic BM25 Search Engine for Teaching and Learning Information Retrieval. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '20), July 25–30, 2020, Virtual Event, China.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3397271.3401413

1 INTRODUCTION

One of the topics discussed at the SIGIR 2019 Open-Source IR Replicability Challenge Workshop (OSIRRC 2019) [7] was the difficulty of writing a search engine. This difficulty was perceived to be one reason that open source was important to the community, and also why some search engine research was conducted outside of the search engine and in packages such as R. Indeed, the open source search engines seen at the workshop were thousands of lines of code in length and complex in their implementations. They were also written in a variety of languages ranging from C/C++ to Java.

However, building a tool to construct and serialise an inverted index need not be complex. Equally, implementing a term-at-a-time (TaaT) search engine can be straightforward given the index.

In this work we set out with the sole objective of writing a simple and straightforward pair of tools for demonstrating this.[1] We present a tool to build and write an inverted index of the TREC Wall Street Journal collection – in under 300 lines of documented code. We also present a tool to search that index, rank using BM25, and produce output compatible with `trec_eval`.[2] The search engine is also under 300 lines of documented code.

This code is a demonstration of "how to do it" in the simplest form. It does not include any of the extras seen in a larger search engine – index compression, stemming, thesaurus, relevance feedback, WAND processing, and so on. We present some of these as possible extensions for those who might wish to use JASSjr as a teaching tool.

2 RELATED WORK

In this section we outline the related work in both open source search engines and in teaching and learning Information Retrieval.

2.1 Open Source

There are many open source search engines, sixteen submissions were seen at OSIRRC 2019,[3] suggesting a healthy open source community in academia, but there are others used in industry.

Several open source search engines are written in Java and are based on the Lucene code base – including Solr and Elastic Search. Both of these are influential in industry where they are used by many organisations. Several academic and experimental search engines are also based on Lucene including the Anserini [6] family: Solrini (based on Solr), Pyserini (the Python interface to Anserini), and Elasterini (Anserini on Elasticsearch). Separate from Anserini there is the ielab submission to OSIRRC 2019. According to Fernández-Luna who experimented with Lucene in a teaching environment [1], "Lucene is a "monster", with lots of classes and methods. This is because it is a large-scale production system, and so students often are frighten [*sic*] by it, unsure of how to work with it". In contrast, the approach we take is to simplify as much as possible in order to clarify how to build an indexer and search engine – which we achieve in under 600 lines of documented code.

IRC-CENTRE2019 [2] is a logistic regression model for document routing. It is not a traditional search engine.

Recent interest in neural IR has seen open source search engines such as Birch [27] and NVSM [26]. Neural models of Information

SIGIR '20, July 25–30, 2020, Virtual Event, China

ACM ISBN 978-1-4503-8016-4/20/07…$15.00
https://doi.org/10.1145/3397271.3401413

[1] Available here: https://github.com/andrewtrotman/JASSjr
[2] https://github.com/usnistgov/trec_eval
[3] The source code for many of the search engines we discuss here is available indirectly through the OSIRRC 2019 GitHub repository at https://osirrc.github.io/osirrc2019/

Retrieval are neither simple nor mature. Consequently we leave it for others to explain this technology.

Terrier [15], also written in Java, is well known within the IR community. It is a scalable search engine used for both research and in industry. It supports many ranking approaches, learning to rank, fast query processing, and so on. It is not a minimalistic search engine for learning.

Galago [8] and Indri [20] are both built on the Lemur toolkit. The latest release of Indri (June 2019) suggests that it is no longer possible to build it and run it with modern tools (MacOS 10.12+, gcc 7+, Visual Studio beyond 2012). Galago is current and continues to be used for research. It supports a complex query language and ranking models – it is not minimalistic.

PISA [16], written in C/C++, is designed as an extensible implementation of state of the art in efficient search. The algorithms it uses include the most recent work on dynamic pruning (MaxScore [25], WAND [3], Block-Max MaxScore [5] and Block-Max WAND [9]). It also includes the most recent work on compression (Partitioned Elias Fano [18]) and SIMD-based compression schemes such as StreamVByte [10] and QMX [24]. Designed to be state of the art, a working knowledge of the academic literature is needed in order to understand the implementation. We make no effort to be state of the art and include neither dynamic pruning nor compression, we focus on simplicity and clarity.

Written in C/C++, the ATIRE [23], JASS [11], JASSv2 [22] sequence of search engines are written to demonstrate the efficiency of impact ordering. ATIRE is comprehensive in its inclusion of compression schemes, ranking functions, stemmers, relevance feedback approaches and so on. JASS, the anytime search engine, demonstrates how to use impact ordering and Score-at-a-Time (SaaT) processing to control server load. JASSv2 is a C++14 re-implementation of ATIRE and written for clarity – but it is still monolithic. None of these three are minimalistic.

The Old Dog search engine [17], like ours, takes a minimalistic approach. The authors suggest that the search engine could rely on an external column-store for the index, and demonstrate this by implementing the ranking function in SQL. While we agree with their general approach (simplicity for understanding), we believe that the requirement for a SQL database as a back-end is over-kill especially if the database already has a built-in search engine. We demonstrate that the index can easily be stored in disk files serialisable without any extra libraries or tools beyond those provided by our programming language.

2.2 Teaching and Learning

Azzopardi et al. [1] describe their hackathon efforts to make Lucene work with TREC collections, and their workshop discussing the use of IR for teaching and learning. At this event Fernández-Luna discussed the University of Granada where teaching included indexing and retrieval models. They initially used SulaIR, a custom built learning environment, but found that students found it difficult to understand both the IR process and how to code it – they later adopted Lucene. Palchowdhury discussed the dangers of treating the search engine as a black box, especially when it has its own implementation of otherwise standard ranking functions. We address these concerns with a minimalistic code base that we believe is

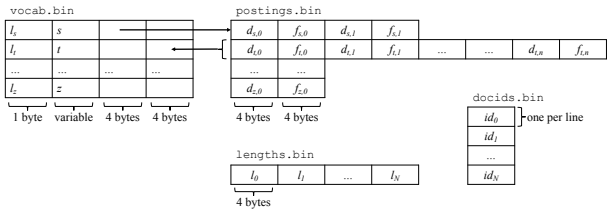

Figure 1: Layout of JASSjr Index.

easy to understand, TREC compatible, and implements the ATIRE version of BM25. Di Buccio and also Balog discussed the contents of their course which includes indexing and searching, and requires students to implement simple systems. We believe our JASSjr could act as a model answer in such courses. Indeed, we initially wrote it as a model answer to our own "write a search engine" exercise.

López-García & Cacheda [13], similar to Fernández-Luna, state that "Although there are a lot of available and mature IR systems on the Internet, their code is often too complex to be explained to students". They go on to introduce IR-Components, a Java framework for making IR tools. Calado et al. [4] introduce the IR-BASE framework for building search components. It is not clear whether IR-Components and IR-Base are available today (2019). What is clear is that research into IR teaching and learning has shown a need for a small and simple example implementation of some of the core technologies in IR – and we provide that for text indexing and search.

3 JASSjr

JASSjr is a straight-forward implementation of an inverted index search engine and its indexer all in under 600 lines of code.

3.1 Indexing

The indexing tool, `JASSjr_index` assumes the TREC file format where each document is stored within a `<DOC>` element, and the document unique identifier (or primary key) is stored in a `<DOCNO>` element within that. In a single pass, and line by line, JASSjr performs lexical analysis and builds an in-memory inverted index. The C/C++ version uses a `std::unordered_map` to store the index, a hash-map from a term to an ordered pair of $< d, f_t >$ (document, term frequency). Primary keys are stored in a `std::vector<std::string>` object, while document lengths (needed for BM25) are stored in a `std::vector<int>` – equivalent structures are used in the Java version. Serialisation involves writing these data structures to disk for later use by the search engine, `JASSjr_search`. The serialised index is document-ordered.

Figure 1 is a schematic representation of the index once serialised to disk. The vocabulary file (`vocab.bin`) is a list of terms and associated data for each term. There are four parts, the first is the length of the term, measured in bytes (and stored in 1-byte). The second is the term itself (stored in a variable number of bytes). Next is a pointer to the start of the postings list for that term, stored as a 4-byte offset from the start of the postings list file, `postings.bin`. The length of the postings list is stored in a 4-byte integer. The Java and C/C++ versions are index-compatible.

The postings file is the concatenation of all the postings lists. Each posting in each postings list consists of two parts, d, the

document number and $f_{t,d}$ the number of time that term t occurs in document d.

From the figure, l_s might be 4, s might be the ('\0' terminated) term "jass", the length of the postings list might be 4 (2 postings at 2 integers each posting). $d_{s,0}$ might be 6, $f_{s,0}$ might be 4, $d_{s,1}$ might be 12, $f_{s,1}$ might be 3 indicating that the term "jass" occurs in document 6 a total of 4 times and in document 12 thrice. Conveniently, as the postings list is not compressed, the document frequency of the term, n_t, can easily be computed from the postings list length.

The length of each document ($\sum_{t \in d} f_{t,d}$) is needed for BM25. In JASSjr, these lengths are serialised in `lengths.bin` as an array of 4-byte integers, one per document. The length of document d, l_d, is accessed by block-reading `lengths.bin` into an array, `length_vector`, and accessing as `length_vector[d]`. JASSjr counts from $d = 0$, the document number of the first document.

The final part of the index, `docids.bin` is a carriage return separated list of the document identifiers (or public keys) as found in the `<DOCNO>` element of the document. That is, it is a text file with one `DOCNO` per line.

3.2 Searching

The search engine uses Term-at-a-Time (TAAT) query processing because it is simpler to implement and understand than the alternatives of Document-at-a-Time (DAAT) seen in Terrier or Score-at-a-Time (SAAT) seen in JASS. The search engine starts by loading the primary keys, document lengths, and vocabulary into memory (but not the postings lists). It then reads a query from `stdin` and processes it term at a time.

This processing starts by initialising an array of accumulators, $\mathbb{A}$, to 0. It continues by loading the postings list for the first term in the query, computing the ranking contribution (the *impact*) of that term to the first document in the postings list, and adds it to that document's accumulator, $\mathbb{A}_d$. It does this for each document in the postings list before moving on to the next – processing the query terms one at a time. In the case of JASSjr, the ranking function is the ATIRE variant of BM25 (which always produces positive scores),

$$\mathbb{A}_d = \sum_{t \in Q} \log(\frac{N}{n_t}) \times \frac{(k_1 + 1) \times f_{t,d}}{k_1 \times ((1 - b) + b \times (\frac{l_d}{L})) + f_{t,d}}, \quad (1)$$

where N is the number of documents in the collection, n_t is the number of documents that contain term t, $f_{t,d}$ is the number of times t occurs in document d, l_d is the length of document d, and L is the average document length. k_1 and b are tuning hyperparameters.

Finally, JASSjr sorts the accumulators from highest to lowest score and the highest at most 1,000 non-zero accumulators are output in `trec_eval` format.

The search engine does not read TREC topic files directly, instead it assumes a more condensed format (which can be used by some other search engines, including ATIRE and JASS). That format is one query per line, where the first token on the line is the query number (required for `trec_eval`). As an example, TREC topics 51 would be "`51 airbus subsidies`".

If the first token in the query is not a number then it is assumed to be part of the query and treated as a searchable term. Our GitHub repository includes TREC topics 51-100 in this format, along with their assessments (the qrel file) as examples.

4 EXTENSIONS

As a minimalistic search engine, there are inumerable projects that might be considered, especially if used in a classroom environment.

The JASSjr index is not compressed. Adding a simple but effective codec such as one from the variable byte family [21] is straightforward. Sample code is readily available in many of the other open source search engines. Adding a novel SIMD-based codec is a substantial amount of work because SIMD codecs are not easy to code. Adding it to JASSjr is straightforward: Encoding of the postings lists could be performed just before serialisation, and decoding immediately after loading from disk.

Adding a stemmer such as Porter [19] involves taking each token from the lexical analyser and stemming it before adding to the in-memory index. It is important to remember to also stem the query terms as they come out of the query lexer otherwise there will be a vocabulary mismatch. A simple classroom exercise might be to implement a simple s-stripping stemmer and to add it to JASSjr.

The indexer does not stop words. There are many readily available stop word lists on the web. Adding stop word support could be done in two ways. Either check each token in each document after the lexical analysis and before adding it to the in-memory index, or check each term in the in-memory index on serialisation.

Everything needed to implement language models [28] with Dirichlet smoothing or with Jelinek-Mercer smoothing are present. Complex rankers such as BM25-adpt [14] require more work as additional postings list processing is required.

The results quality could be improved by adding a Learning-to-Rank post-filter. This might be done using a NeuralIR model. The efficiency could be improved using WAND or other early termination algorithms. Functionality could be increased with phrase searching and fielded search. These extensions are beyond minimalistic model we adopt.

The interface to JASSjr is text-based. We strongly encourage experimentation in user interfaces both on the web and off. An HTML interface could be straightforward. A tool to load the documents given their primary key might be required as an extension.

5 EVALUATION

In this section we compare JASSjr with two other open source search engines (ATIRE and JASS). We demonstrate that the indexing process is slower (but reasonable), that the index size is larger (but not unreasonable), that search time is longer (but acceptable), and that the quality of the results is near that of other search engines. We do not evaluate it as a teaching tool because we have an insufficient number of participants to do so.

Our experiments were conducted on an otherwise idle Mac running macOS Mojave 10.14.5 on an Intel Core i5 at 3.2Ghz with 24GB 1867 MHz DDR3 RAM using the Apple LLVM version 10.0.1 (clang-1001.0.46.4) with maximum optimisations, and Java 12.0.2. We compare JASSjr to ATIRE and JASS simply because we have familiarity with them. This is not intended to be a comprehensive evaluation of open source search engines, which others have done [12] and continue to do [6], but we observe that ATIRE and JASS have performed well in prior evaluation exercises.

We use the TREC WSJ collection of 173,252 documents and totalling 508.5MB in size. Testing was with the titles of TREC topics

	C/C++	Java	ATIRE	JASS
Indexing Time	18.47s	25.29s	8.96s	14.25s
Search Time (all 50)	1.0688s	2.2666	0.2826s	0.1698s
Index Size	326MB	326MB	67MB	284MB
MAP (from `trec_eval`)	0.2080	0.2080	0.2128	0.2116

Table 1: Mean time in seconds to index and search the TREC WSJ document collection of 173,252 documents using TREC topic 51-100 titles. Index size in MB (1024*1024 bytes), and mean average precision at top *k=1000*.

51-100. This evaluation is intended to be indicative of the performance on a *reasonable* sized document collection that might be used in a learning environment.

Time was measured using the `real` time from the macOS `time` command. Reported numbers are mean times over 5 repetitions. Timings include start-up, search, and the time to output the top $k = 1,000$ documents in `trec_eval` format. The quality of the results was measures using `trec_eval`, but only mean average precision (MAP) is reported. The BM25 hyperparameters were set at the same values for each search engine – the ATIRE defaults.

We make no comment on the efficiency of C/C++ over Java, but the results in Table 1 show our C/C++ implementation being faster than our Java one. We discuss the C/C++ version henceforth.

Indexing time for JASSjr is longer than the others, but only by a factor of 1.3. The ATIRE indexer uses a sophisticated parallel indexing pipeline and so is the fastest. JASS stores impact ordered indexes and so must compute the impact scores for each term in each document while indexing – which takes time.

The search time for JASSjr is longer that that of both ATIRE and JASS, taking 4 times as long as ATIRE and 6 times longer than JASS. Both JASS and ATIRE have accumulator management strategies along with top-k management during search time – which substantially reduce search time. JASSjr uses an array of accumulators and sorts it using `qsort()` – neither of which is efficient.

The ATIRE index is the smallest at 67MB. By default ATIRE uses variable byte compression for the postings lists and a number of other techniques for decreasing the index size (including short postings list management). JASS uses Group Elias Gamma SIMD encoding for the postings and no compression for the vocabulary or the pointers to the postings lists. JASSjr does not compress.

The results quality (from `trec_eval`) are comparable. ATIRE and JASSjr use the same ranking function but different lexical analysers – JASSjr uses alphanumeric tokens but ATIRE separates alphas from numerics. JASS and ATIRE differ in quality because JASS uses pre-computed impact scores quantised into 255 buckets, whereas ATIRE computes BM25 on the fly (also contributing to some of the search time). We did not perform statistical significance tests as the purpose of reporting MAP is to demonstrate comparable performance is comparable, not that one is better than other others.

Overall, we believe that JASSjr performs well when compared to ATIRE and JASS. There is clearly room for improvement, but that is partly the point – these improvements might be used as a learning exercise as many take few lines of code.

6 CONCLUSIONS

In this work we have demonstrated that it is possible to write a `trec_eval` compatible search engine that ranks using BM25 in fewer than 600 lines of documented C/C++ or Java code. We have made our code available online and with the BSD 2-clause licence. An explanation of the indexing and the searching process is given, and the serialised index is presented in detail. We hope that others will use this work in their classrooms and extend it. We plan to add a Python implementation soon.

REFERENCES

[1] L Azzopardi, Y Moshfeghi, M Halvey, R S Alkhawaldeh, K Balog, E Di Buccio, D Ceccarelli, J M Fernández-Luna, C Hull, J Mannix, and S Palchowdhury. 2017. Lucene4IR: Developing Information Retrieval Evaluation Resources Using Lucene. *SIGIR Forum* 50, 2 (2017), 58–75.

[2] T Breuer and P Schaer. 2019. Dockerizing Automatic Routing Runs for The Open-Source IR Replicability Challenge (OSIRRC 2019). In *OSIRRC2019*. 31–35.

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

[4] P Calado, A Cardoso-Cachopo, and A L Oliveira. 2007. IR-BASE: An Integrated Framework for the Research and Teaching of Information Retrieval Technologies. In *TLIR2007*. 2–2.

[5] K Chakrabarti, S Chaudhuri, and V Ganti. 2011. Interval-based Pruning for Top-k Processing over Compressed Lists. In *ICDE2011*. 709–720.

[6] R Clancy, T Eskildsen, N Ruest, and J Lin. 2019. Solr Integration in the Anserini Information Retrieval Toolkit. In *SIGIR2019*. 1285–1288.

[7] R Clancy, N Ferro, C Hauff, T Sakai, and Z Z Wu (Eds.). 2019. *Proceedings of the Open-Source IR Replicability Challenge (OSIRRC 2019)*. Vol. 2409. CEUR-WS.org.

[8] B Croft, D Metzler, and T Strohman. 2009. *Search Engines: Information Retrieval in Practice* (1st ed.). Addison-Wesley Publishing Company.

[9] S Ding and T Suel. 2011. Faster Top-k Document Retrieval Using Block-max Indexes. In *SIGIR 2011*. 993–1002.

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

[11] J Lin and A Trotman. 2015. Anytime Ranking for Impact-Ordered Indexes. In *ICTIR2015*. 301–304.

[12] J 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 *ECIR2016*. 408–420.

[13] R López-García and F Cacheda. 2011. A Technical Approach to Information Retrieval Pedagogy. The Information Retrieval Series, Vol. 31. Springer, 89–105.

[14] Y Lv and C Zhai. 2011. Adaptive Term Frequency Normalization for BM25. In *CIKM2011*. 1985–1988.

[15] C Macdonald, R McCreadie, R L T Santos, and I Ounis. 2012. From puppy to maturity: Experiences in developing Terrier. *OSIR2012*.

[16] A Mallia, M Siedlaczek, J Mackenzie, and T Suel. 2019. PISA: Performant Indexes and Search for Academia. In *OSIRRC 2019*.

[17] H Mühleisen, T Samar, J Lin, and A de Vries. 2014. Old Dogs Are Great at New Tricks: Column Stores for Ir Prototyping. In *SIGIR2014*. 863–866.

[18] G Ottaviano and R Venturini. 2014. Partitioned Elias-Fano Indexes. In *SIGIR2014*. 273–282.

[19] Martin F Porter. 1980. An algorithm for suffix stripping. *Program* 14 (1980), 130–137.

[20] T Strohman, D Metzler, H Turtle, and W B Croft. 2005. Indri: A language model-based search engine for complex queries. In *Proceedings of the International Conference on Intelligent Analysis*.

[21] A Trotman. 2014. Compression, SIMD, and Postings Lists. In *ADCS 2014*. 50:50–50:57.

[22] A Trotman and M Crane. 2019. Micro- and macro-optimizations of SaaT search. *SP&E* 49, 5 (2019), 942–950.

[23] A Trotman, X-F Jia, and M Crane. 2012. Towards an Efficient and Effective Search Engine.

[24] A Trotman and J Lin. 2016. In Vacuo and In Situ Evaluation of SIMD Codecs. In *ADCS 2016*. 1–8.

[25] H Turtle and J Flood. 1995. Query Evaluation: Strategies and Optimizations. *IP&M* 31, 6 (1995), 831–850.

[26] C Van Gysel, M de Rijke, and E Kanoulas. 2018. Neural Vector Spaces for Unsupervised Information Retrieval. *TOIS* 36, 4, Article 38 (2018).

[27] Z A Yilmaz, W Yang, H Zhang, and J Lin. 2019. Cross-Domain Modeling of Sentence-Level Evidence for Document Retrieval. In *EMNLP-IJCNLP*.

[28] C Zhai and J Lafferty. 2004. A Study of Smoothing Methods for Language Models Applied to Information Retrieval. *TOIS* 22, 2 (2004), 179–214.