A Common Framework for Exploring Document-at-a-Time and Score-at-a-Time Retrieval Methods

Andrew Trotman
Department of Computer Science
University of Otago
Dunedin, New Zealand

Joel Mackenzie
School of Information Technology and Electrical Engineering
The University of Queensland
Brisbane, Australia

Pradeesh Parameswaran
Department of Computer Science
University of Otago
Dunedin, New Zealand

Jimmy Lin
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Canada

ABSTRACT

Document-at-a-time (DaaT) and score-at-a-time (SaaT) query evaluation techniques are different approaches to top-k retrieval with inverted indexes. While modern systems are dominated by DaaT, the academic literature has seen decades of debate about the merits of each. Recently, there has been renewed interest in SaaT methods for learned sparse lexical models, where studies have shown that transformers generate "wacky weights" that appear to reduce opportunities for optimizations in DaaT methods. However, researchers currently lack an easy-to-use SaaT system to support further exploration. This is the gap that our work fills. Starting with a modern SaaT system (JASS), we built Python bindings in order to integrate into the DaaT Pyserini IR toolkit (Lucene). The result is a common frontend to both a DaaT and a SaaT system. We demonstrate how recent experiments with a wide range of learned sparse lexical models can be easily reproduced. Our contribution is a framework that enables future research comparing DaaT and SaaT methods in the context of modern neural retrieval models.

CCS CONCEPTS

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

KEYWORDS

Anserini, JASS, Python, Efficiency, Procrastination

ACM Reference Format:
Andrew Trotman, Joel Mackenzie, Pradeesh Parameswaran, and Jimmy Lin. 2022. A Common Framework for Exploring Document-at-a-Time and Score-at-a-Time Retrieval Methods. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '22), July 11–15, 2022, Madrid, Spain.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3477495.3531657

SIGIR '22, July 11–15, 2022, Madrid, Spain

ACM ISBN 978-1-4503-8732-3/22/07…$15.00
https://doi.org/10.1145/3477495.3531657

1 INTRODUCTION

Despite rapid advances in neural ranking models, primarily driven in the past few years by pretrained transformer models, the humble decades-old inverted index and similarly old "traditional" bag-of-words query evaluation techniques remain indispensable, primarily for two reasons:

(1) A large class of neural models, generally known as cross encoders, operate as rerankers over initial candidates generated by a first-stage retriever [2, 18, 22]. Furthermore, fusion methods that combine scores derived from transformer-based representations with a bag-of-words scoring method such as BM25 generally outperform either alone [11, 18, 21, 34].
(2) Researchers have had much success with a class of models known as learned sparse models, which generate lexical (i.e., bag-of-words) representations with weights that are derived from transformers. That is, neural models are trained to generate term weights that maximize some retrieval objective given large amounts of supervised training data. These term weights can be viewed as impact scores [1], and retrieval with these models can be directly performed using inverted indexes.

At a high level, query evaluation techniques for the top-k retrieval problem with inverted indexes can be divided into document-at-a-time (DaaT) and score-at-a-time (SaaT) methods. The advantages and disadvantages of each have been the subject of academic debate for decades [1, 5, 7, 33], although for practical purposes, DaaT methods dominate production systems today. For example, the open-source Lucene search library, perhaps the most widely deployed search platform today, uses DaaT methods [10].

However, SaaT methods are drawing renewed interest, at least by the academic community. Recently, Mackenzie et al. [24] demonstrated that, in the context of learned sparse lexical models, SaaT methods have certain appealing characteristics. They can be faster than DaaT methods under certain settings, and exhibit far less variability in query evaluation latency (e.g., much greater control for so-called "tail latency") [5]. Furthermore, they can have smaller indexes, and overall their implementations are simpler.

The research community currently lacks an easy-to-use toolkit to further explore DaaT and SaaT methods, especially in the context of modern neural models. Our work fills this gap via a common Python-based frontend that integrates both Lucene (a DaaT system)

and JASS (a SaaT system) in the Pyserini IR toolkit [16] through PyJASS, a Python wrapper for JASS.

As a demonstration of these capabilities, this integration allows the Lucene and JASS experiments presented by Mackenzie et al. [24] to be reproduced with ease (whereas the original paper required working with disparate systems). Our common framework enables future explorations of DaaT and SaaT methods.

2 DEMONSTRATION

We begin "at the end" to demonstrate the final result of our efforts. Consider uniCOIL [15], a recent learned sparse lexical model. With the Pyserini IR toolkit [16], reproducing a run on the popular MS MARCO passage ranking task with the development queries using Lucene can be accomplished with a single command:

```
python -m pyserini.search.lucene \
  --index msmarco-passage-unicoil-d2q \
  --topics msmarco-passage-dev-subset-unicoil \
  --output runs/run.msmarco-passage.unicoil.tsv \
  --output-format msmarco \
  --batch 36 --threads 12 \
  --hits 1000 \
  --impact
```

Most of the options are self explanatory, but a few key features are worth explicitly pointing out. The value of the `--index` option, `msmarco-passage-unicoil-d2q`, refers to a pre-built index that Pyserini will download from a known location the first time the command is invoked; the index will be cached locally for subsequent reuse. Similarly, the pre-tokenized queries are also hosted at a known location, and will be downloaded and cached. Pyserini itself is published as an artifact on PyPI, which means that everything needed to reproduce this run is self-contained. Once the `pyserini` package is installed, the user does not need anything else. Batteries are included.

The above capability already exists as described by Lin et al. [16]. However, until now, Pyserini depended on Lucene as its "retrieval backend". The contribution of this work is that we have integrated JASS as another backend alongside Lucene. Thus, to perform the same run with JASS, we can issue the following command:

```
python -m pyserini.search.jass \
  --index jass-msmarco-passage-unicoil-d2q \
  --topics msmarco-passage-dev-subset-unicoil \
  --output runs/run.msmarco-passage.unicoil.tsv \
  --output-format msmarco \
  --batch 36 --threads 12 \
  --hits 1000 \
  --impact
```

The main distinction here is that instead of calling the main driver program in `pyserini.search.lucene`, we call `pyserini.search.jass` instead. In other words, we have a consistent frontend for both a DaaT system (Lucene) and a SaaT system (JASS).[1]

In concrete terms, what "win" does this capability create for researchers and practitioners in our community? For those simply "using" learned sparse lexical models as the first-stage retriever in a multi-stage ranking architecture, Pyserini supports seamless switching between different backends. As our experiments later show, using the JASS backend is much faster, so users get a boost in speed "for free". For researchers interested in further exploring DaaT vs. SaaT methods in the context of modern learned sparse lexical representations, a uniform frontend streamlines experimental practice and enables fair "apples to apples" comparisons.

For both use cases, Python is a key selling point, since Python has emerged as the *lingua franca* of IR research today in the context of neural models since all the major deep learning toolkits (PyTorch and Tensorflow) have adopted Python as the frontend language. The Pyserini driver programs above in turn call wrapper objects around Lucene and JASS, and both provide easy-to-use APIs that support deeper code integration.

3 IMPLEMENTATION

Having described the "final product" of our efforts, we turn our attention to the implementation details. We start with an overview of Pyserini as it existed prior to this work. JASS was integrated in two steps: As it was written in C++, we first built PyJASS, which provides a Python wrapper for JASS. To align APIs and the actual frontend (described in the previous section), we had to write further wrapper classes around PyJASS in Pyserini. These details are described below.

3.1 Pyserini

Anserini [36, 37] began as an effort to better align the research and practice of building search applications via the creation of a research-focused IR toolkit around the open-source Lucene search library, which lies at the core of widely deployed platforms such as Solr and Elasticsearch. Pyserini [16] began as a lightweight Python wrapper to Anserini for the Python-centric world of deep learning (as Anserini, like Lucene, was implemented in Java). However, it has since evolved into a toolkit for reproducible information retrieval research with sparse and dense representations, integrating support for dense retrieval models [14, 18, 20, 35] via Facebook's Faiss library [13]. Thus, Lucene and Faiss form the two primary backends in Pyserini, for retrieval with sparse (i.e., bag-of-words) and dense representations, respectively.

Taking another step towards the goal of better aligning academia and industry, recent work has demonstrated seamless replicability and interoperability between Elasticsearch and Pyserini [6]. For academic researchers, such efforts illustrate the potential real-world impact of laboratory innovations by providing a plausible translation path to real-world deployments. For industry practitioners, this integration provides access to evaluation resources and infrastructure built by the academic community. This two-way exchange is mutually beneficial, and the efforts described here add JASS as another backend alongside Lucene, further expanding the scope of potential academic–industrial exchanges.

3.2 JASS

As far as we are aware, JASS (v2) [19, 31] is the only actively maintained open-source SaaT search engine available today. It is written in C++ 17 and runs on Mac and Linux. At present, a modern Intel CPU is required because, internally, JASS uses SIMD-enhanced

[1] Note that both Lucene and JASS also have system-specific parameters which can be set at search time.

Elias Gamma compression [32], which in turn relies on AVX2 instructions (or AVX-512 instructions if available).

SaaT search engines rely on an indexing phase in which the contribution of a term to a document—its impact score—is computed and stored in the index as a small integer. A consequence of this is that the ranking function at retrieval (search) time boils down to integer addition. But unlike DaaT, partial scores must be stored for each document that might be in the top-k. These are traditionally stored in an array of accumulators that spans the entire document space, which can make SaaT slow.

As a solution to this challenge, JASS uses a page-table like structure to only allocate and initialize accumulators that are at or near the documents scored during index traversal. In prior work [12], this innovation has been shown to be one of the main reasons why JASS can process the postings so quickly—it doesn't allocate or initialize accumulators that aren't used.

As an anytime search engine, the tradeoff between effectiveness and efficiency in JASS is controlled using ρ, a parameter that specifies the proportion of the postings that should be processed [19, 23]. Previous experiments have found that setting ρ to a fixed value corresponding to 10% of the number of documents in the collection represents a "sweet spot" for balancing efficiency and effectiveness [19].

3.3 PyJASS

As a standalone search engine, JASS of course has its own frontend, which was used to produce the experimental results reported by Mackenzie et al. [24]. However, the point of our work is to enable seamless integration, and to that end, the first step was to produce Python bindings for JASS. PyJASS provides Python bindings for JASS, and is available as an artifact on the Python Package Index (PyPI). That is, PyJASS can be used independently as a Python interface to JASS, but it is the integration with Pyserini (more below) that provides the common frontend discussed in Section 2.

Producing bindings from C++ to any other language can be laborious, but fortunately there are several tools that help automate this process. We chose SWIG [4], as we have used it in the past for other projects. Using SWIG, it is straightforward to write the bindings interface, and the tool is generally well supported. Indeed, our SWIG interface file simply includes a bunch of C++ header files we were already using for the C++ APIs.

To generate the Python-specific bindings we invoke SWIG configured to produce Python bindings, and then compile JASS and the SWIG-generated wrapper into a shared object. At the end of the build process we obtain two files: `pyjass.py` and `_pyjass.so`, which comprise the Python interface itself and JASS in a library, respectively. With additional packaging, PyJASS is available on PyPI, and can be installed with standard tools such as `pip`. For ease of installation, we recommend using the Anaconda package manager, and our documentation provides instructions for doing so.

3.4 Final Integration

While it is possible, in principle, to integrate JASS *directly* in Pyserini, the approach we took allows PyJASS to stand independently as a self-contained search engine. We can think of several scenarios in which this would be useful, for example, to aid further development of JASS by providing a lightweight, scriptable mechanism for debugging and analysis.

However, to enable seamless integration, we needed to write additional Python wrapper classes to bring the PyJASS API and Pyserini API into alignment. In Pyserini, `pyserini.search.lucene` is a "main" driver program that manages the query lifecycle, dispatching to the `LuceneSearcher` class, which is itself a wrapper around functionalities implemented in Java that access Lucene directly. The searcher object has a `search` method and a `batch_search` method (for multi-threaded searching), along with many other convenience and utility methods. The driver program manages the entire process of generating a standard TREC run: reading topic files to extract queries, sending queries to the searcher object, gathering results, and writing the ranked lists to a final run file.

The implementation in `pyserini.search.jass` follows exactly the same design. We have implemented the `JASSv2Searcher` class with essentially the same API, managed by a main driver class in the same way as with Lucene. The final product is seamless integration, supporting the one-command reproduction examples presented in Section 2.

3.5 Additional Tooling

Our integration efforts have focused on a common frontend for querying. At present, JASS and Pyserini maintain separate indexing pipelines. Pyserini (via Anserini) builds indexes using Lucene, and provides support for a multitude of document formats (TREC format, JSON, WARCs, etc.). Similarly, JASS has its own indexer that supports TREC formats and CIFF, the Common Index File Format [17]. It is worth noting that indexing pipelines and additional tooling around manipulating corpora are largely "out of scope" with respect to the goals we hope to achieve here. In truth, it is neither desirable nor possible to build a common framework for these capabilities. We explain below.

By definition, DaaT and SaaT systems require different index organizations, and any integration effort will necessarily be superficial (e.g., wrappers). At least for researchers, indexing is a one-time event since work is focused on relatively few common test collections, for which we already provide pre-built indexes. Thus, many researchers may never need to touch an indexer.

Furthermore, building the inverted indexes is merely the final stage in a long sequence of corpus preparation steps that start with a neural retrieval model. Obviously, how to train these models is beyond the scope of our work. However, even given a model, the researcher next needs to encode the entire corpus—i.e., generate the representations that correspond to each document. While there are some commonalities, models can also be idiosyncratic. Furthermore, encoding text into vector representations requires GPUs in practice, which brings in its own infrastructure requirements.

Thus, we currently believe that attempts to integrate data preparation and indexing are not worthwhile. Nevertheless, data abstractions remain helpful: For example, we have standardized on a corpus representation for learned sparse lexical retrieval models. Since they are, for each document, a mapping from tokens to weights, a JSON-based interchange format is straightforward. In our current implementation, both the Pyserini and JASS indexers take corpora

in this format as input, and are agnostic with respect to how these representations are generated. Furthermore, other exchange tools and formats, such as the Common Index File Format [17], can be used to facilitate easy index interchange.

4 EXPERIMENTS

The primary demonstration of the capabilities that we've built is easy reproduction of the results of Mackenzie et al. [24], along the lines of the self-contained Pyserini commands shown in Section 2. In contrast, the original paper required working with three disparate systems: this work brings Lucene and JASS under a common frontend. Mackenzie et al. [24] explored a third DaaT system, PISA [26], which lies beyond the scope of these integration efforts, and thus we do not present those results here. We begin by summarizing the experimental conditions explored in that work and then present experimental results.

4.1 Setup and Configurations

Experiments used the MS MARCO passage corpus [3], which comprises 8.8M passages drawn from paragraph-length extracts generated by Bing's question answering module. On the 6980 queries in the development set, we evaluated the following models:

BM25 [30] over bag-of-words representations of the corpus passages, with $k_1 = 0.82$ and $b = 0.68$, following Lin et al. [16].

BM25 w/ doc2query–T5 (BM25-T5 for short) represents a document expansion condition where the corpus is augmented with predictions generated by doc2query [27, 28] using the T5 [29] neural sequence-to-sequence model. Ranking is still performed using BM25 scoring (i.e., does not involve neural inference).

DeepImpact [25] is an example of a retrieval model based on learned sparse lexical representations. Candidate terms in each document that should be assigned non-zero weights by the transformer model are generated from doc2query–T5.

uniCOIL + doc2query–T5 [15] (uniCOIL-T5 for short) is a simplification of the COIL model [9]. Instead of generating vector weights, uniCOIL produces scalar weights, using doc2query–T5 as the source for candidate terms (like DeepImpact).

uniCOIL + TILDE [38] (uniCOIL-TILDE for short) replaces candidate term selection using doc2query–T5 with an alternative model based on TILDE [39].

SPLADEv2 [8] learns sparse lexical representations of documents using the masked language model (MLM) head in BERT. One advantage is that using the MLM head replaces the need for an explicit expansion step, compared to, for example, using doc2query–T5.

All models in our experiments are based on the implementations described by Mackenzie et al. [24]. JASS indexes are built using CIFF extracts [17] from Lucene, so the indexes contain exactly the same information. For the learned sparse lexical models, we use the "pseudo-document trick" to take advantage of existing indexing pipelines. For each document (from the JSON corpus format described in Section 3.5), the indexer creates a "fake document" where a term is repeated the same number of times as its (quantized) integer weight assigned by the neural model. This value is stored in the term frequency position of a standard inverted index. For retrieval, we simply use a scoring function that computes the sum of term frequencies. Note that we perform search using pre-tokenized queries; query inference (with the neural models) is conducted offline.

We report results for retrieving the top $k = 1000$ documents, which would be suitable for use as a first-stage retriever in a multi-stage ranking pipeline. All experiments were conducted in memory on a Linux machine with two 3.50 GHz Intel Xeon Gold 6144 CPUs and 512 GiB of RAM. We evaluate both single thread and multi-thread performance, detailed below; the latter implementation is based on intra-query parallelism with query batches.

	Lucene	**JASS**	Speedup
BM25	66	13	5.1×
BM25-T5	101	33	3.1×
DeepImpact	209	51	4.1×
uniCOIL-T5	211	150	1.4×
uniCOIL-TILDE	193	83	2.3×
SPLADEv2	1042	308	3.4×
Total	1822	638	2.9×

Table 1: Total end-to-end running time (in seconds) for processing the development queries of the MS MARCO passage ranking task, with 32 threads and a batch size of 100.

4.2 Results

To illustrate the usefulness of our common interface, we report results on two sets of experiments.

Table 1 presents our first experiment, which shows the benefit of using JASS as the retrieval backend within Pyserini for accelerated experimentation, focused on query throughput. Here, a hypothetical researcher is interested in evaluating the six models described above on the MS MARCO queries as quickly as possible, using all available processing power; on our test machine, this translates into running experiments on 32 threads and a batch size of 100. The table shows the end-to-end running time of `pyserini.search.lucene` and `pyserini.search.jass`, which includes the time for reading indexes and queries, computing the rankings, and writing the output files. The final column notes the speedup of JASS relative to Lucene. In total, across all experiments, we observe that JASS is approximately three times faster than Lucene.

Our second experiment demonstrates the use of our common interface for experimenting with effectiveness/efficiency tradeoffs. Table 2 shows these results. For each model, we report output quality, measured in terms of mean reciprocal rank at cutoff 10 (RR@10), the official metric of the test collection, query latency in milliseconds, and index size measured in megabytes. We also report the mean latency from the "base" version of both Anserini (Java) and JASS (C++) to quantify the overheads caused by the Python bindings. Quality and space figures are identical to a similar table in Mackenzie et al. [24] (obviously, since we are reproducing those results), but latency figures are different; we find that the Python bindings cause some overhead in both cases, but are more pronounced with the Lucene backend. With the PyJASS backend, search takes only a fraction of a millisecond longer than the underlying C++ implementation.

	Quality	Time (ms)		Space
Method	RR@10	Base	Python	MiB
Anserini (Lucene): DaaT				
BM25	0.187	38.8	46.1	661
BM25-T5	0.277	62.6	71.5	1036
DeepImpact	0.325	235.4	248.4	1417
uniCOIL-T5	0.352	209.7	215.8	1313
uniCOIL-TILDE	0.350	171.4	183.6	2067
SPLADEv2	0.369	2087.5	2068.8	4987
JASS: SaaT				
BM25	0.187	7.9	8.0	729
BM25-T5	0.277	28.9	29.0	947
DeepImpact	0.326	22.4	22.4	1354
uniCOIL-T5	0.352	96.1	96.2	1139
uniCOIL-TILDE	0.350	53.9	53.9	1782
SPLADEv2	0.369	219.3	219.5	3595

Table 2: Experimental results on the development queries of the MS MARCO passage ranking test collection. Latency is reported as the mean response time using a single thread.

Confirming the results of Mackenzie et al. [24], we observe JASS to be substantially faster than Lucene, with both approaches yielding the same effectiveness. But this isn't exactly a fair comparison because Lucene provides full-featured, production-ready search infrastructure, while JASS is a research system with little in the way of bells and whistles. This is a bit like comparing the speed of a family sedan with a stock race car. With the former, you can safely transport your family in comfort. The latter doesn't have cupholders or even doors that open. The main point of this exercise is to demonstrate that we have successfully built a common frontend to support future explorations of DaaT and SaaT methods.

5 CONCLUSIONS

The work reported here represents an instance of our broader efforts to build easy-to-use tools in support of reproducible research in information retrieval. Modern IR research, with its heavy emphasis on neural models, has evolved into a complex mishmash of different toolkits and environments. However, the Python language appears to be the glue that holds everything together, even when the underlying implementations are written in C++ and Java. Pyserini builds on this foundation and leverages standard tooling (e.g., PyPI packages) and best practices to the extent possible, with the aim of "reducing friction" for IR researchers. The demonstration described here is a step towards our broader vision. Looking again at Mackenzie et al. [24], the obvious next step would be to integrate the PISA search engine [26], which will get us closer to "the one frontend to rule them all". We look forward to pursuing this in future work.

ACKNOWLEDGEMENTS

This research was partially supported by the Australian Research Council Discovery Project DP200103136 and the Natural Sciences and Engineering Research Council (NSERC) of Canada.

REFERENCES

[1] Vo Ngoc Anh, Owen de Kretser, and Alistair Moffat. 2001. Vector-Space Ranking with Effective Early Termination. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2001)*. 35–42.

[2] Nima Asadi and Jimmy Lin. 2013. Effectiveness/Efficiency Tradeoffs for Candidate Generation in Multi-Stage Retrieval Architectures. In *Proceedings of the 36th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2013)*. Dublin, Ireland, 997–1000.

[3] Payal Bajaj, Daniel Campos, Nick Craswell, Li Deng, Jianfeng Gao, Xiaodong Liu, Rangan Majumder, Andrew McNamara, Bhaskar Mitra, Tri Nguyen, Mir Rosenberg, Xia Song, Alina Stoica, Saurabh Tiwary, and Tong Wang. 2018. MS MARCO: A Human Generated MAchine Reading COmprehension Dataset. *arXiv:1611.09268v3* (2018).

[4] David M. Beazley. 2003. Automated Scientific Software Scripting With SWIG. *Future Generation Computer Systems* 19, 5 (2003), 599–609.

[5] Matt Crane, J. Shane Culpepper, Jimmy Lin, Joel Mackenzie, and Andrew Trotman. 2017. A Comparison of Document-at-a-Time and Score-at-a-Time Query Evaluation. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining (WSDM 2017)*. 201–210.

[6] Josh Devins, Julie Tibshirani, and Jimmy Lin. 2022. Aligning the Research and Practice of Building Search Applications: Elasticsearch and Pyserini. In *Proceedings of the 15th ACM International Conference on Web Search and Data Mining (WSDM 2022)*. 1573–1576.

[7] Marcus Fontoura, Vanja Josifovski, Jinhui Liu, Srihari Venkatesan, Xiangfei Zhu, and Jason Zien. 2011. Evaluation Strategies for Top-k Queries over Memory-Resident Inverted Indexes. *Proc. VLDB Endow.* 4, 12 (2011), 1213–1224.

[8] Thibault Formal, Carlos Lassance, Benjamin Piwowarski, and Stéphane Clinchant. 2021. SPLADE v2: Sparse Lexical and Expansion Model for Information Retrieval. *arXiv:2109.10086* (2021).

[9] Luyu Gao, Zhuyun Dai, and Jamie Callan. 2021. COIL: Revisit Exact Lexical Match in Information Retrieval with Contextualized Inverted List. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Online, 3030–3042.

[10] Adrien Grand, Robert Muir, Jim Ferenczi, and Jimmy Lin. 2020. From MaxScore to Block-Max WAND: The Story of How Lucene Significantly Improved Query Evaluation Performance. In *Proceedings of the 42nd European Conference on Information Retrieval, Part II (ECIR 2020)*. 20–27.

[11] Sebastian Hofstätter, Sheng-Chieh Lin, Jheng-Hong Yang, Jimmy Lin, and Allan Hanbury. 2021. Efficiently Teaching an Effective Dense Retriever with Balanced Topic Aware Sampling. In *Proceedings of the 44th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2021)*. 113–122.

[12] Xiang-Fei Jia, Andrew Trotman, and Richard O'Keefe. 2010. Efficient accumulator initialisation. In *Proceedings of the 15th Australasian Document Computing Symposium*. 44–51.

[13] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2021), 535–547.

[14] Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Online, 6769–6781.

[15] Jimmy Lin and Xueguang Ma. 2021. A Few Brief Notes on DeepImpact, COIL, and a Conceptual Framework for Information Retrieval Techniques. *arXiv:2106.14807* (2021).

[16] Jimmy Lin, Xueguang Ma, Sheng-Chieh Lin, Jheng-Hong Yang, Ronak Pradeep, and Rodrigo Nogueira. 2021. Pyserini: A Python Toolkit for Reproducible Information Retrieval Research with Sparse and Dense Representations. In *Proceedings of the 44th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2021)*. 2356–2362.

[17] Jimmy Lin, Joel Mackenzie, Chris Kamphuis, Craig Macdonald, Antonio Mallia, Michał Siedlaczek, Andrew Trotman, and Arjen de Vries. 2020. Supporting Interoperability Between Open-Source Search Engines with the Common Index File Format. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2020)*. 2149–2152.

[18] Jimmy Lin, Rodrigo Nogueira, and Andrew Yates. 2021. *Pretrained Transformers for Text Ranking: BERT and Beyond.* Morgan & Claypool Publishers.

[19] Jimmy Lin and Andrew Trotman. 2015. Anytime Ranking for Impact-Ordered Indexes. In *Proceedings of the ACM International Conference on the Theory of Information Retrieval (ICTIR 2015)*. 301–304.

[20] Sheng-Chieh Lin, Jheng-Hong Yang, and Jimmy Lin. 2021. In-Batch Negatives for Knowledge Distillation with Tightly-Coupled Teachers for Dense Retrieval. In *Proceedings of the 6th Workshop on Representation Learning for NLP (RepL4NLP-2021)*. 163–173.

[21] Xueguang Ma, Kai Sun, Ronak Pradeep, Minghan Li, and Jimmy Lin. 2022. Comparing Score Aggregation Approaches for Document Retrieval with Pretrained

Transformers. In *Proceedings of the 44th European Conference on Information Retrieval (ECIR 2022), Part I*. Stavanger, Norway, 613–626.

[22] Joel Mackenzie, J. Shane Culpepper, Roi Blanco, Matt Crane, Charles L. A. Clarke, and Jimmy Lin. 2018. Query driven algorithm selection in early stage retrieval. In *Proceedings of the 11th ACM International Conference on Web Search and Data Mining (WSDM 2018)*. 396–404.

[23] Joel Mackenzie, Falk Scholer, and J. Shane Culpepper. 2017. Early Termination Heuristics for Score-at-a-Time Index Traversal. In *Proceedings of the 22nd Australasian Document Computing Symposium (ADCS 2017)*. 8.1–8.8.

[24] Joel Mackenzie, Andrew Trotman, and Jimmy Lin. 2021. Wacky Weights in Learned Sparse Representations and the Revenge of Score-at-a-Time Query Evaluation. *arXiv:2110.11540* (2021).

[25] Antonio Mallia, Omar Khattab, Torsten Suel, and Nicola Tonellotto. 2021. Learning Passage Impacts for Inverted Indexes. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2021)*. 1723–1727.

[26] Antonio Mallia, Michał Siedlaczek, Joel Mackenzie, and Torsten Suel. 2019. PISA: Performant Indexes and Search for Academia. In *Proceedings of the Open-Source IR Replicability Challenge (OSIRRC 2019): CEUR Workshop Proceedings Vol-2409*. 50–56.

[27] Rodrigo Nogueira and Jimmy Lin. 2019. From doc2query to docTTTTTquery.

[28] Rodrigo Nogueira, Wei Yang, Jimmy Lin, and Kyunghyun Cho. 2019. Document Expansion by Query Prediction. *arXiv:1904.08375* (2019).

[29] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67.

[30] Stephen Robertson and Hugo Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Foundations and Trends in Information Retrieval* 3, 4 (2009), 333–389.

[31] Andrew Trotman and Matt Crane. 2019. Micro- and Macro-optimizations of SaaT Search. *Software: Practice and Experience* 49, 5 (2019), 942–950.

[32] Andrew Trotman and Kat Lilly. 2018. Elias Revisited: Group Elias SIMD Coding. In *Proceedings of the 23rd Australasian Document Computing Symposium (ADCS 2018)*. Article 4, 8 pages.

[33] Howard R. Turtle and James Flood. 1995. Query Evaluation: Strategies and Optimizations. *Information Processing & Management* 31, 6 (1995), 831–850.

[34] Shuai Wang, Shengyao Zhuang, and Guido Zuccon. 2021. BERT-Based Dense Retrievers Require Interpolation with BM25 for Effective Passage Retrieval. In *Proceedings of the 44th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2021)*. 317–324.

[35] Lee Xiong, Chenyan Xiong, Ye Li, Kwok-Fung Tang, Jialin Liu, Paul N. Bennett, Junaid Ahmed, and Arnold Overwijk. 2021. Approximate Nearest Neighbor Negative Contrastive Learning for Dense Text Retrieval. In *Proceedings of the 9th International Conference on Learning Representations (ICLR 2021)*.

[36] Peilin Yang, Hui Fang, and Jimmy Lin. 2017. Anserini: Enabling the Use of Lucene for Information Retrieval Research. In *Proceedings of the 40th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2017)*. Tokyo, Japan, 1253–1256.

[37] Peilin Yang, Hui Fang, and Jimmy Lin. 2018. Anserini: Reproducible Ranking Baselines Using Lucene. *Journal of Data and Information Quality* 10, 4 (2018), Article 16.

[38] Shengyao Zhuang and Guido Zuccon. 2021. Fast Passage Re-ranking with Contextualized Exact Term Matching and Efficient Passage Expansion. *arXiv:2108.08513* (2021).

[39] Shengyao Zhuang and Guido Zuccon. 2021. TILDE: Term Independent Likelihood moDEl for Passage Re-ranking. In *Proceedings of the 44th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2021)*. 1483–1492.