Insights into the Efficiency of Open-Source Score-at-a-Time Search Engines: A Reproducibility Study

Katelyn Harlan
harka424@student.otago.ac.nz
University of Otago
Dunedin, New Zealand

Andrew Trotman
andrew.trotman@otago.ac.nz
University of Otago
Dunedin, New Zealand

Veronica Liesaputra
veronica.liesaputra@otago.ac.nz
University of Otago
Dunedin, New Zealand

Abstract

Score-at-a-Time (SaaT) retrieval, utilising impact-ordered indexes, remains a relatively overlooked search strategy with increasing importance. Within the context of learned sparse representations and approximate retrieval, SaaT has proven to be a competitive alternative to the popular Document-at-a-Time (DaaT) approach. Yet, there are only two notable implementations of SaaT search engines: JASSv2 and IOQP. We are interested in the differences between these systems and how those differences affect performance. We identify differences in postings lists due to indexing, ranking functions, and quantization. Thus, we introduce ciffTools for quantizing the CIFF indexes used by both search engines; eliminating these differences. Then, in a reproducibility study we reproduce previous experiments investigating the efficiency gap between the systems. They also differ in their compression codecs, with IOQP using SIMD BP-128 and JASSv2 using Elias Gamma SIMD VB. We, unexpectedly, find in another reproducibility experiment that SIMD BP-128 and QMX outperform Elias Gamma SIMD VB *in situ*. Finally, we investigate the CPU effect, and find the engines are affected differently. Overall, JASSv2 has faster median latency on a server-grade CPU, while on a desktop-grade they are evenly matched. IOQP has faster tail latency regardless of CPU. Our work reduces the throughput difference between JASSv2 and IOQP and offers insights into which aspects affect the efficiency of SaaT.

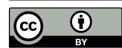
CCS Concepts

• **Information systems → Retrieval efficiency**; *Search index compression.*

Keywords

Score-at-a-Time Retrieval; Impact-Ordered Indexes; Index Compression; Learned Sparse Retrieval; Reproducibility

ACM Reference Format:
Katelyn Harlan, Andrew Trotman, and Veronica Liesaputra. 2026. Insights into the Efficiency of Open-Source Score-at-a-Time Search Engines: A Reproducibility Study. In *Proceedings of the 49th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '26), July 20–24, 2026, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3805712.3808554



This work is licensed under a Creative Commons Attribution 4.0 International License.
SIGIR '26, Melbourne, VIC, Australia
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2599-9/2026/07
https://doi.org/10.1145/3805712.3808554

1 Introduction

For efficient information retrieval, we still rely on the inverted index [26, 28]. Usually, this index will maintain, for each unique term, a list of documents containing the term (the postings list), often with additional information, e.g. the total occurrences of each term in each document, or a pre-computed impact score. In Score-at-a-Time (SaaT) retrieval, these indexes are impact-ordered. Rather than sorting on document ID, each postings list is organised into segments — where each segment contains a list of documents with the same impact scores.

In learned sparse retrieval (LSR), neural networks are used to learn the term-document weights. Mackenzie et al. [31] remark that the weight distributions produced by these networks are "somewhat unusual": they found there were reduced opportunities for pruning and early termination. Accordingly, rather than using the more traditional Document-at-a-Time (DaaT) approach to retrieval, it appears that SaaT retrieval can be more efficient for LSR [31, 32].

Though much of the current landscape is concerned with dense retrieval [20, 36], it is clear that sparse retrieval, particularly with learned sparse representations, is increasingly relevant. We highlight two recent methods for retrieval with LSR, both found to significantly outperform other existing models: SEISMIC [7], and Block-Max Pruning (BMP) [35]. Hybrid retrieval [4, 9, 47], combining dense and sparse methods, is also popular. Indeed, it seems to be more effective than dense or sparse retrieval individually. So, any improvements or discoveries made regarding sparse retrieval will also have an impact on hybrid retrieval. However, we remark that LSR is not always appropriate; well-labelled datasets are not always available. Additionally, though there are many recent works on approximate retrieval, such as SEISMIC, exhaustive retrieval is still worth consideration.

JASSv2 and IOQP are the only well-known open-source SaaT search engines. They are capable of both approximate and exhaustive query processing, and are not limited to learned sparse representations. In an effort to gain insights into the efficiency of these two systems, so that we may further improve upon SaaT retrieval, we reproduce the work of Mackenzie et al. [28] — performing efficiency and effectiveness comparisons between JASSv2 and IOQP. We confirm previous results that IOQP outperforms JASSv2. For clarity, our setup is mostly different from previous works, though we do use much of the same source code.

Alongside this reproducibility study, we provide a resource to facilitate working with the CIFF [25] format which both JASSv2 and IOQP use for indexing. In our initial attempts to recreate the experiments by Mackenzie et al. [28], we were unable to produce consistent postings lists between JASSv2 and IOQP, which led to vastly different results. In order to ensure a fair comparison it is

necessary to use the same index, and that is the purpose of the CIFF format — to ensure the documents are processed and ordered consistently. However, to our knowledge, there is no readily available tool to quantize a CIFF for use in impact-ordered systems. Thus, we introduce our open-source ciffTools — which eliminates differences in ranking caused by different ranking functions and quantization methods. Further details on this resource are in Section 3.

Though there are many differences in the two search engines, detailed in Section 2.2, we take particular note of the compression of the postings lists as compression is known to have a large effect on the throughput of in-memory search engines [27]. We reproduce further experiments in JASSv2 and IOQP using various compression codecs, and find that we are able to minimise the gap in performance. Importantly, we note that we end up with a different conclusion to past findings by Trotman [39] and Trotman & Lilly [42]. Perhaps this is due to developments in computer architectures over the last ten years, or improvements in compiler vectorization. Indeed, Trotman [39] found the performance of a codec to be platform specific. Our experiments are also *in situ*, and use a greater range of collections and models.

Finally, we provide a brief investigation into the effects of using different hardware. We run all experiments on a second machine and find that the query latency is dependent on both the search engine and the CPU architecture (other than frequency).

Our main contributions in this work are as follows:

- We introduce our open-source ciffTools to extend the use of CIFF beyond document-ordered indexes to include impact-ordered indexes. (Section 3)
- We reproduce previous experiments comparing the efficiency and effectiveness of JASSv2 and IOQP, and confirm that IOQP is more efficient (with default codecs). (Section 5)
- We present a novel comparison of the efficiency of JASSv2 and IOQP with different integer compression codecs, and find we can reduce latency for both systems. (Section 6)
- We provide a brief investigation into the effects of different hardware, and find that query latency is affected by the CPU architecture differently in each search engine. (Section 7)
- We provide QMX in Rust and SIMD BP-128 in C++ through foreign function interfaces.

Our initial findings indicate that IOQP generally outperforms JASSv2. However, by changing the compression codec, we can improve the performance of both engines whilst also reducing the efficiency gap. Specifically, we found that both SIMD BP-128 and QMX are faster than Elias Gamma SIMD VB. Additionally, we find that IOQP and JASSv2 are differently affected by the hardware, suggesting accumulator management may be sensitive to the CPU architecture (other than frequency). Our experiments show that for median latency, JASSv2 outperforms IOQP on the server-grade CPU, while they are evenly matched on the desktop-grade. IOQP has faster tail latency regardless of CPU.

2 Background

2.1 Score-at-a-Time / Impact-Ordered Indexes

The inverted index is a staple of modern information retrieval. For each unique term, the index maintains a list of the documents it appears in with some additional information, usually term frequency. Such a list is referred to as a postings list and is of the form: $< d_1, f_{t,d_1} >, < d_2, f_{t,d_2} >, \ldots, < d_n, f_{t,d_n} >$, where d_n is the document ID and f_{t,d_n} is the term frequency for term t in document d_n. Specifically, this is a document-ordered index.

Rather than storing term frequencies, what if we stored some pre-computed score? Ranking functions such as BM25 [38] are given as a sum of the contributions of each query term. Thus, if we pre-compute each term-document contribution, we can simply sum them at query time. Indeed, we refer to these pre-computed weights as impact scores. Most ranking models used to create these scores produce floating-point numbers, which are cumbersome to compress. Thus, the impacts are quantized. This is normally done by uniformly quantizing the full range of impacts into b bits, resulting in integer scores in the range $[0, 2^b - 1]$ [1]. Further, if we organise the document IDs into segments corresponding to each impact score, we can sort the postings list by impact: $< i_t : d_1, d_2, \ldots, d_n >$, where i_t is an impact score corresponding to term t and $d_1, \ldots, d_n$ are the document IDs within the segment. This is used in an impact-ordered index [1, 26].

In Score-at-a-Time retrieval [1], a query is processed in decreasing impact order. This allows for effective results, even with early termination. The process begins by determining which segments to process, and in which order. That is, the postings lists for all query terms are found, and the segments ordered by impact score. These segments are then processed in this order, and the top-k documents identified. Clearly, this search paradigm benefits from the structure of an impact-ordered index.

2.2 JASSv2 & IOQP

JASSv2 [26, 40][1] is a SaaT search engine in C++. Although it is equipped to index some TREC collections, it is largely designed to index from a CIFF as it is more focused on efficient/effective search than indexing.

Currently, the default compression scheme for the postings lists is Elias Gamma SIMD VB [42], henceforth referred to as EG VB.

For accumulator management, JASSv2 uses a slightly modified version of the approach by Jia et al. [18]. The accumulator array, containing one 16-bit accumulator per document, is modelled as a two-dimensional array. This is then broken into a series of pages, and a dirty-flag assigned for each page. Before writing to a particular accumulator, the dirty-flag is checked. The page is only zeroed if the flag is set, after which the flag is cleared and the accumulator updated. Otherwise, it is known to be clean. The dirty-flags are set to 1 at start of search, but the pages are only initialised when needed. In JASSv2, the page size is always a whole power of 2.

During query processing, JASSv2 maintains a heap of the top-k documents. When the current impact is added to the accumulator, it is checked against the smallest score in the heap and, if it is greater, a pointer to the accumulator is added. This means that the algorithm is interruptible. For approximate query processing, i.e. early termination, JASSv2 will process up to — but no more than — ρ postings. If evaluating the next segment would put ρ over, it will terminate. As segments are never partially processed, and segments

[1] https://github.com/andrewtrotman/JASSv2 (commit ef17edf)

with lower impacts are typically longer, the algorithm will become more accurate as the time budget increases [26].

IOQP [28][2], implemented in Rust, is another SaaT search engine. It entirely relies on CIFF for indexing. SIMD BP-128 [22, 23] and StreamVByte [24] are used for integer compression. Rather than employing an accumulator management strategy as in JASSv2, IOQP will simply zero the accumulator table at the start of each query — relying on the Rust compiler to optimize the process [28].

Unlike JASSv2, IOQP does not keep track of the top-k documents during query processing. Instead, for each chunk of accumulators, a maximum-score is maintained. Once processing is terminated, the first k accumulators are pushed into the min-heap, establishing an entry threshold. Then, only chunks with a maximum score greater than the current threshold are visited. Once all the chunks have either been visited or skipped, the heap will contain the top-k documents. Importantly, we note that early termination also differs to JASSv2. That is, IOQP will always process at least ρ postings; if the total postings processed is under ρ, it will still process the next segment even if it would result in processing more than ρ postings. In our experiments, we modified this behaviour so that it is consistent with JASSv2.

2.3 Integer Compression & Postings Lists

Postings lists, whether impact-ordered or not, consist of document IDs, i.e. integers. These are typically stored as compressed delta-gaps (d-gaps), where the differences between document IDs are used rather than the document IDs themselves. This allows for better compression as it leads to smaller integers [42]. The choice of compression codec used for the postings list has an impact on the efficiency of search, and the effectiveness of indexing (i.e. compression ratio, or index size). In our experiments, we encounter the following codecs:

Elias Gamma SIMD VB [42] — In Elias Gamma [15], integers are processed one-by-one and encoded in two parts. First, the base-2 magnitude $M = \lfloor \log_2(x) \rfloor$ is stored in unary, then the integer itself stored in binary.

In Elias Gamma SIMD VB, this is extended to work with 512-bit SIMD words. First, the 512-bit word (i.e. payload) is broken into 16 × 32-bit "rows". Then, 16 integers are read, the base-2 magnitude (M) of the largest is computed, and this is written in unary to a 32-bit selector. From each row, a column of M bits is allocated and the integers striped across them. This process continues until the payload is full. The encoded sequence is stored with each 32-bit selector followed by its corresponding 512-bit payload. We remark that the result will be larger than with non-SIMD Elias Gamma encoding, as the fixed-width columns result in wasted bits, but that the decoding will be much faster due to the use of SIMD instructions.

Shorter lists use variable byte encoding as implemented in JASSv2; integers are broken into 7-bit chunks and stored big endian, with the high bit of each chunk acting as a termination flag.

StreamVByte [24] — StreamVByte is a variant of variable byte encoding that takes advantage of SIMD instructions. Accordingly, integers are considered in blocks of 4. To keep track of the size of each integer, we assign a control byte to each block — with each individual byte length using 2 bits. Then, since the number of compressed integers is recorded (N), the first $\lceil \frac{2N}{8} \rceil$ bytes of the encoded sequence can be used to store them — followed by the compressed integers. .

SIMD BP-128 [22, 23] — In binary packing, integers are grouped into blocks and packed as tightly as possible in a variable number of words, with the bit width (i.e. the number of bits needed to represent the largest number in binary) stored as an extra byte. For SIMD BP-128, blocks consist of 128 integers. For every 16 such blocks, a 16-byte selector (of 16 × 1-byte bit widths) is stored before the 16 blocks of 128 integers. The integers in each block are packed to the width of the largest integer in said block.

Of course, there are usually leftover blocks. These are handled separately. In each case, a selector byte containing the bit width is written before each block of 128 encoded integers. Finally, any remaining integers (less than 128) are compressed using a different codec altogether. We use StreamVByte, though any form of variable byte encoding is suitable.

QMX [39, 43] — Unlike the above codecs, QMX (Quantities Multipliers eXtractors) targets both long and short postings lists. The idea is to pack as many integers as possible into a single 128-bit word, with the integers in each word being stored in the same number of bits. These payloads are stored consecutively, with corresponding bit widths specified by selectors stored after the encoded sequence. Each selector is a single byte, where the first 4 bits describe the bit packing, and the rest give a run-length. Originally, QMX stored a variable byte encoded pointer (corresponding to the start of the selectors) at the end of the encoded sequence. The implementation we use, based on experiments by Trotman and Lin [43], removes this pointer. Instead, the selectors are stored (and traversed) in reverse sequence, i.e. the final byte of the encoded sequence is the first selector, and the second to last byte is the second.

2.4 Related Work

There is much recent work pertaining to dense retrieval [20, 36], and though it does overcome the classic term mismatch problem it is not without its limitations. For one, it is computationally expensive to train dense models, and often an appropriate dataset is not available for training. Further, dense models are not nearly as effective on out-of-domain datasets, whereas sparse models are much more robust [4, 9, 21]. To that end, we see many hybrid approaches, which combine dense and sparse methods [4, 9, 47]. Indeed, many tasks use a sparse model for a first ranking, followed by a dense model [2]. Thus, the quality of sparse retrieval has a direct impact on the overall ranking.

Learned sparse representations, which use LLMs to encode input into sparse embeddings, are seeing a rise in popularity. With this, and the incorporation of sparse methods into deep retrieval pipelines, we are seeing much work on improving the efficiency of sparse retrieval [5, 6, 32, 34, 35]. There is also an effort to improve the effectiveness of the sparse embeddings themselves [16, 17], though this is not the focus of our work.

Block-Max Pruning [35] (BMP) and SEISMIC [7] are two recent approaches to learned sparse retrieval. BMP is a dynamic pruning strategy operating in two main phases: block filtering, and evaluation. In the first phase, they divide the document IDs into blocks, noting the maximum impact score for each block — these scores

[2] https://github.com/JMMackenzie/IOQP (commit 00aff0e)

are later used to calculate upper bounds when processing queries. In the second phase, blocks are processed until a termination condition is met. These blocks are accessed in decreasing order of the upper bounds. For this strategy, a hybrid between an inverted and forward index was used, and a top-k heap used for search. SEISMIC is a method for specifically approximate LSR retrieval. It also uses an interesting design that employs both inverted and forward indexes. Inverted lists are organised into blocks, each containing a summary vector that allows for blocks to be easily skipped. When a summary instead indicates that a block should be examined, the forward index is used to retrieve the exact embeddings. Essentially, SEISMIC introduces an exciting approach to early termination.

It has been shown that SEISMIC outperforms IOQP [7]. Thus, it will also outperform JASSv2. However, we point out that SEISMIC only uses approximate query processing, whereas we also examine exhaustive query processing. Further, SEISMIC is not compatible with CIFF, and only works with embeddings. That is, SEISMIC is created for approximate retrieval over learned sparse representations. Similarly, BMP also outperforms IOQP [35]. Though BMP does use CIFF indexes, and can perform exhaustive processing, it is designed specifically for LSR.

3 ciffTools

The Common Index File Format (CIFF) [25], used by both JASSv2 and IOQP, is a format for sharing indexes between search systems. CIFF eliminates differences in rankings caused by different document processing pipelines, and ensures the documents are consistently ordered. However, it does not support comparisons between SaaT search engines with impact-ordered indexes; it does not eliminate differences caused by ranking functions or quantization methods.

As with variable byte encoding, referring to BM25 [38] can be quite inconsistent due to the many variants [19, 44]. Yet, it has been found that the different implementations of BM25 result in no discernible difference in overall effectiveness [19, 44]. Lin et al. [25] found that even so, there were still differences in ranking on a per-topic basis, and that the use of their CIFF format greatly reduced such differences. But what of efficiency? If different ranking functions are used — whether BM25 or something else entirely — it will influence the impact scores, and result in inconsistencies between postings lists — particularly when combined with quantization. Thus, it is still important to ensure the same ranking function is used.

Perhaps even more crucial, is the method of quantization used. Even with the same ranking function and using the same number of bits, a difference in methodology will result in different postings lists — and thus a fair efficiency comparison cannot be made, as it would involve comparing different indexes.

JASSv2 and IOQP both use ATIRE BM25 [41], but slightly different means of quantization. Thus, even starting from the same CIFF, they produce vastly different results — which we encountered in our preliminary comparisons of the two engines. To facilitate a fairer comparison we eliminated differences in ranking, and thus we present our open-source resource: ciffTools[3]. This takes a CIFF and quantizes it by converting term frequencies into impact scores; eliminating any discrepancies caused by different ranking functions and quantization methods.

[3]https://github.com/Axiomatic314/ciffTools

For ranking, as in JASSv2 and IOQP, we use ATIRE BM25:

$$r_q = \sum_{t \in q} \log\left(\frac{N}{df_t}\right) \cdot \frac{(k_1 + 1) \cdot tf_{t,d}}{k_1(1 - b + b \cdot (\frac{L_d}{L_{avg}})) + tf_{t,d}}$$

Where N is the number of documents in the collection, df_t is the number of documents containing term t, $tf_{t,d}$ is the corresponding frequency of the term in document d, and L_d and L_{avg} are the length of the document d and the average length of all documents, respectively. Finally, the parameters k_1 and b are tunable.

However, JASSv2 and IOQP use different implementations of uniform quantization [1]. Given as:

JASSv2:

$$i_t = \left\lfloor \frac{x_{t,d} - L}{U - L} \cdot 2^b \right\rfloor + 1$$

IOQP:

$$i_t = \left\lceil x_{t,d} \cdot \frac{2^b}{U} \right\rceil$$

where i_t is the quantized impact corresponding to document d and term t, $x_{t,d}$ is the corresponding impact, b is the number of bits to quantize the impact into, and U and L are the highest and lowest impact scores in the collection, respectively.

Importantly, we see that these will give a different range of integers. JASSv2 outputs integers in the range $[1, 2^b + 1]$, and IOQP $[0, 2^b]$. Thus, we use a slightly modified version of uniform quantization [1]. Our implementation ensures impact scores are kept in the range $[1, 2^b - 1]$ — avoiding overflow while ensuring all terms have a contribution. This is defined as:

$$i_t = \left\lfloor \left((2^b - 2) \cdot \frac{x_{t,d} - L}{U - L}\right) + 1 \right\rfloor$$

We have also included a secondary function of producing a human-readable dump of a CIFF, quantized or not.

To demonstrate the benefits of using our solution, let us consider a small example: the query “New Fuel Sources” on the TREC Robust04 collection. Without using ciffTools, we find that the top 10 results are entirely different between JASSv2 and IOQP (even considering tiebreakers). In particular, the document at rank 1 for JASSv2 is at rank 10 for IOQP. With the use of ciffTools, the results become more consistent. The top 10 results for both search engines contain the same documents, and any differences are due to tiebreaking.

4 Experimental Setup

4.1 The Pipeline

We use a variety of tools to prepare our indexes for our experiments: Anserini [46], CIFF [25], ciffTools, and Faster Graph Bisection (FGB) [30]. With the exception of ciffTools, we follow the pipeline used by Mackenzie et al. [28].

Anserini is a Java software toolkit built around the open source search engine Lucene. It focuses on reproducibility, with many guides for working with standard IR collections — including the majority of those used in our experiments.

CIFF is a binary data exchange format for indexes using protocol buffers. Provided in Java, the namesake repo[4] takes in indexes

[4]https://github.com/osirrc/ciff/tree/master

produced by Anserini, and gives a CIFF index that can be ingested by a compatible search engine (e.g. JASSv2 or IOQP).

Our ciffTools, as detailed in Section 3, is a collection of Go tools for working with the CIFF format. Notably, it provides a means of producing quantized CIFF indexes — ensuring that scoring and quantization remains consistent between search engines.

FGB expands on the bipartite graph partitioning by Dhulipala et al. [14], making several changes to reduce computation time and workload. This tool, offered in Rust, topically reorders the documents in a CIFF, which results in smaller indexes and faster query throughput [29].

Our indexing process is as follows:

(1) Index with Anserini. We use the optimize flag to combine into a single index segment for CIFF compatibility. It is also crucial to use Anserini's DumpAnalyzedQueries tool to apply consistent processing (e.g. stemming) on the queries.
(2) Convert the Anserini index into a CIFF.
(3) Quantize the CIFF with ciffTools.
(4) Reorder the quantized CIFF using FGB. We used the default gain function.
(5) Input the reordered, quantized CIFF into JASSv2/IOQP.

We note that in the experiments by Mackenzie et al. [28], Step 3 was not used. Indeed, to our knowledge, there is no readily available tool to produce a quantized CIFF, other than our ciffTools.

4.2 Document Collections

Following the work by Mackenzie et al. [28], we utilise the Gov2 corpus and the MSMARCO passage collection in our experiments. We further extend these comparisons by including the Robust04 collection, allowing us to see the efficiency of these systems with small collections in addition to the large corpus (Gov2) and sparse indexes (MSMARCO).

Robust04 is a small collection of 522, 565 documents from TREC. Queries are from the TREC 2004 robust track [45], and effectiveness is measured in MAP. We use the default ATIRE BM25 values of $k_1 = 0.9$ and $b = 0.4$.

Gov2 contains 25 million documents from a crawl of the .gov domain. We use topics 701–850 from the TREC terabyte tracks [8, 10, 11], with MAP as our effectiveness measure. Again, we use ATIRE BM25 $k_1 = 0.9$ and $b = 0.4$

Finally, we use the MSMARCO passage collection, consisting of 8.8 million passages. We use a mix of traditional and neural-augmented models: ATIRE BM25 on the original index (BM25), ATIRE BM25 on a doc2query-T5 expanded index (BM25-T5) [37], and ATIRE BM25 on a DeepCT weighted index (DeepCT) [12, 13]. As well as some learned sparse models: DeepImpact [33], uniCOIL-TILDE [48], and SPLADEv2 [16]. To keep our experiments consistent with prior work, we use the same ATIRE BM25 parameters as Mackenzie et al. [28]. Aside from DeepCT, the MSMARCO collections used $k_1 = 0.82$ and $b = 0.68$. As the parameters for DeepCT were not given, we followed the recommendations from the DeepCT project[5]: $k_1 = 10$ and $b = 0.9$. We used the dev queries [3] for all models, reporting effectiveness as RR@10.

[5]https://github.com/AdeDZY/DeepCT

4.3 General Implementation Details / Setup

All experiments were conducted single-threaded with in-memory indexes on an otherwise idle machine running Linux Mint 21 and equipped with an Intel i7-9800X (3.80 GHz) and 128GB RAM. We ran each experiment 9 times to a depth of $k = 1000$. The mean, median, and tail latencies were taken over each set of 9 runs. Results were obtained for both exhaustive and approximate query processing. In the case of approximate processing, we set ρ to 10% of the collection size. We used 8-bit quantization to produce our indexes, and 16-bit accumulators for search. Further, we note that timing for JASSv2 was modified to ignore the cost of parsing query terms, to stay consistent with the timing for IOQP, and thus the work done by Mackenzie et al. [28]. Finally, we implemented JASSv2 termination logic into IOQP — so both engines will process at most ρ postings.

Throughout this project we used rustc 1.87.0-nightly (2025-03-26) to compile our Rust, and g++ 12.3.0 to compile our C++. We used -O3 optimizations.

5 JASSv2 vs IOQP

In this experiment, we reproduce the efficiency and effectiveness experiments done by Mackenzie et al. [28] on JASSv2 and IOQP. For each collection we report mean, median, and 99th percentile latency, along with the appropriate effectiveness measure. Note that throughout this section, as with the previous study, there is no distinguishable difference in effectiveness between the two search engines.

In Table 1, we present the results of searching on the Robust04 collection with exhaustive and approximate query processing. Mean, median, and tail latency are nearly indistinguishable on this scale.

Table 2 shows the efficiency for Gov2. For mean and median latency there is little difference between the two, though JASSv2 has noticeably higher tail latency.

Finally, in Table 3, we observe the efficiency across the various MSMARCO models. For both exhaustive and approximate processing, IOQP completely outperforms JASSv2.

To determine the statistical significance of our results, we employed the t-test for independent samples (not assuming equal variance) and used a threshold of 0.05. We found that IOQP showed a statistically significant improvement over JASSv2 across all the vast majority of both exhaustive and approximate processing. The exceptions being mean and median latency for approximate Gov2 and exhaustive MSMARCO-DeepCT.

These results are as expected. The trends we observe with both Gov2 and MSMARCO are mostly consistent with the findings of Mackenzie et al. [28] across both exhaustive and approximate processing, with one exception — approximate BM25-T5 on MSMARCO. In the aforementioned work, they discovered uncharacteristically high tail latency for IOQP on this collection, even suggesting IOQP may benefit from adopting JASSv2 termination logic [28]. Indeed, we found that this considerably improved IOQP's performance.

6 Compression

In Section 2.2, we highlighted several differences between JASSv2 and IOQP — notably integer compression codecs, and accumulator management strategies. In the interest of finding the main cause of difference in performance, we focus now on compression — as

Table 1: Mean, median, and 99th percentile latency (ms) and MAP scores for JASSv2 and IOQP on the Robust04 collection. Percentage improvements over JASSv2 are given in parentheses. † indicates statistically significant improvement over other scores.

Model	JASSv2				IOQP			
	Mean	P_{50}	P_{99}	MAP	Mean	P_{50}	P_{99}	MAP
Exhaustive								
BM25	0.45	0.43	1.19	**0.253**	**0.38† (15.4)**	**0.38† (12.9)**	**0.86† (27.7)**	**0.253 (0.0)**
Approximate								
BM25	0.38	0.42	0.60	**0.249**	**0.34† (10.7)**	**0.38† (9.6)**	**0.48† (19.1)**	**0.249 (0.0)**

Table 2: Mean, median, and 99th percentile latency (ms) and MAP scores for JASSv2 and IOQP on the Gov2 collection. Percentage improvements over JASSv2 are given in parentheses. † indicates statistically significant improvement over other scores.

Model	JASSv2				IOQP			
	Mean	P_{50}	P_{99}	MAP	Mean	P_{50}	P_{99}	MAP
Exhaustive								
BM25	15.3	11.8	52.7	**0.305**	**14.0† (8.5)**	**11.6† (2.2)**	**40.1† (23.8)**	**0.305 (0.0)**
Approximate								
BM25	**9.2**	**10.1†**	25.8	**0.297**	**9.2** (−0.1)	10.2 (−1.3)	**14.6† (43.4)**	**0.297 (0.0)**

Table 3: Mean, median, and 99th percentile latency (ms) and RR@10 scores for JASSv2 and IOQP on the MSMARCO passage collection. Percentage improvements over JASSv2 are given in parentheses. † indicates statistically significant improvement over other scores.

Model	JASSv2				IOQP			
	Mean	P_{50}	P_{99}	RR	Mean	P_{50}	P_{99}	RR
Exhaustive								
BM25	8.2	6.7	28.2	**0.188**	**6.6† (19.6)**	**5.5† (18.0)**	**20.6† (26.8)**	**0.188** (−0.1)
BM25-T5	33.4	18.7	481.4	**0.274**	**16.4† (50.8)**	**15.8† (15.5)**	**44.4† (90.8)**	**0.274** (0.0)
DeepCT	3.2	2.9	9.1	0.243	3.1 (1.6)	2.8 (2.3)	**8.1† (10.5)**	**0.244** (0.0)
DeepImpact	23.6	25.0	63.7	**0.327**	**18.0† (23.9)**	**18.4† (26.6)**	**49.3† (22.6)**	**0.327** (0.0)
uniCOIL-TILDE	58.2	48.7	197.3	**0.350**	**44.8† (23.1)**	**37.5† (22.9)**	**151.1† (23.4)**	**0.350** (0.0)
SPLADEv2	231.2	227.9	443.5	**0.369**	**185.9† (19.6)**	**182.8† (19.8)**	**359.0† (19.1)**	0.368 (0.2)
Approximate								
BM25	5.7	6.4	8.2	**0.187**	**5.0† (12.2)**	**5.6† (13.4)**	**7.0† (13.7)**	**0.187** (−0.1)
BM25-T5	4.8	4.9	18.1	0.272	**3.8† (19.6)**	**4.0† (17.2)**	**6.9† (61.6)**	**0.273** (0.1)
DeepCT	3.2	2.9	7.8	**0.243**	**3.0† (4.4)**	**2.8† (4.6)**	**6.8† (12.9)**	**0.243** (0.0)
DeepImpact	6.1	6.5	8.3	**0.319**	**5.2† (15.9)**	**5.4† (16.9)**	**7.1† (14.9)**	0.318 (−0.1)
uniCOIL-TILDE	7.2	7.3	8.8	0.335	**6.4† (11.3)**	**6.4† (12.4)**	**8.3† (6.0)**	**0.336** (0.2)
SPLADEv2	7.7	7.7	9.2	**0.319**	**6.5† (15.2)**	**6.5† (15.6)**	**8.2† (11.2)**	0.318 (−0.5)

it is known to have a large effect on throughput [27]. To that end, we perform a comparison on the efficiency of JASSv2 and IOQP with different integer compression codecs. In doing so, we partially reproduce previous work by Trotman [39] and Trotman & Lily [42]. Note we use the same experimental setup as in Section 5.

Trotman [39] performed comparisons between QMX and various SIMD codecs, namely SIMD BP-128. He found that QMX was more efficient and more space effective than SIMD BP-128. Trotman & Lilly [42] extended the Elias algorithms to benefit from SIMD instructions, with comparisons between Elias Gamma SIMD VB and QMX. They found that Elias Gamma SIMD VB was more efficient, but less effective than QMX.

In our experiments, we recreate both of these comparisons. We compare the performance of Elias Gamma SIMD VB, QMX, and SIMD BP-128 within JASSv2, and QMX and SIMD BP-128 within IOQP. By replicating these experiments *in situ* — on more modern hardware, across two different search engines, and across more document collections/models — we hope to gain some new insights regarding their performance.

By default, JASSv2 uses Elias Gamma SIMD VB, though previously the default was QMX [26, 40]. IOQP defaults to SIMD BP-128

Table 4: Index Sizes for JASSv2 and IOQP with different postings list compression. Reported in MB. Percentage differences to EGVB are given in parentheses.

Model	JASSv2			IOQP	
	EG VB	QMX	BP-128	QMX	BP-128
Robust04					
BM25	**199**	212 (−0.07)	204 (−0.03)	254 (−0.28)	246 (−0.24)
Gov2					
BM25	**9054**	9287 (−0.03)	9451 (−0.04)	11331 (−0.25)	11503 (−0.27)
MSMARCO					
BM25	688	696 (−0.01)	**679** (0.01)	845 (−0.23)	828 (−0.20)
BM25-T5	**907**	931 (−0.03)	934 (−0.03)	1101 (−0.21)	1104 (−0.22)
DeepCT	494	500 (−0.01)	**478** (0.03)	621 (−0.26)	599 (−0.21)
DeepImpact	1422	1435 (−0.01)	**1392** (0.02)	1643 (−0.16)	1600 (−0.13)
uniCOIL-TILDE	**1869**	1950 (−0.04)	1936 (−0.04)	2032 (−0.09)	2019 (−0.08)
SPLADEv2	**3770**	3959 (−0.05)	4008 (−0.06)	4043 (−0.07)	4092 (−0.09)

(and StreamVByte on smaller blocks). Accordingly, we used the same experimental setup as in Section 5, but with the addition of JASSv2 using SIMD BP-128, and IOQP using QMX. We did not put Elias Gamma SIMD VB in IOQP, as we found QMX to be more worthwhile in our preliminary experiments.

For consistency, we used the original implementations of each algorithm from each search engine, i.e. rather than implementing QMX in Rust by hand, or SIMD BP-128 and StreamVByte in C++, we used zero-overhead gateways in the form of foreign function interfaces. Both Rust and C++ compile to native, and are linked together into the final executables.

We will briefly touch on the space effectiveness of the various compression schemes, demonstrated by their corresponding index sizes seen in Table 4. For the JASSv2 indexes, we see that all three codecs are reasonably close over all of the collections. The biggest difference between sizes is on the Gov2 collection with about 400MB between Elias Gamma SIMD VB and SIMD BP-128. We observe a similar trend for the IOQP indexes, again noting that the largest difference in size is over Gov2 (about 200MB). Finally, we note that all of the JASSv2 indexes are smaller than their IOQP counterparts.

To be consistent with prior work, we previously reported mean, median, and 99th percentile latency as well as effectiveness for our search results. For clarity, we only report median and 99th percentile latency. These are more indicative of performance than the mean latency, and the effectiveness is unaffected by the codecs.

We begin with Table 5, which shows the results of searching on the Robust04 collection for exhaustive and approximate query processing. Overall, the performance between the two search engines, and between the different compression schemes was very similar — almost unnoticeable on the scale of milliseconds.

Moving to our largest collection, Table 6 contains the results of searching on Gov2. Here, we see some more variation in performance. For median latency, JASSv2 (particularly with BP-128) performed better than IOQP for both exhaustive and approximate processing. As for tail latency, IOQP (particularly with BP-128) outperformed JASSv2 for approximate processing.

Finally, in Table 7, we examine the results of searching on the MS-MARCO passage collection across various ranking models. For exhaustive and approximate processing, JASSv2 and IOQP are largely similar. The exception being exhaustive BM25-T5, where JASSv2 has much larger tail latency regardless of codec. We also notice that within each search engine, there is not much of a difference in performance between QMX and SIMD BP-128.

To determine if any of our results were statistically significant, we performed the Tukey HSD test with a p-value of 0.05. Our intention was to determine, for each collection and query mode, the best JASSv2 codec and then the best IOQP codec, to then compare with each other. However, for JASSv2, we found that neither SIMD BP-128 nor QMX were better overall — though they were better than Elias Gamma VB. For IOQP, while we did find that QMX was generally better for tail latency, we could not make the same call for median latency. Thus, our JASSv2 vs IOQP comparisons were performed across all four variations with SIMD BP-128 and QMX. For median latency, we found that JASSv2 and IOQP were evenly matched. Though we were unable to determine if QMX or SIMD BP-128 was better as it differed between collections. Interestingly, we remark that IOQP typically performed better with QMX (as taken from JASSv2) rather than its included codecs. For tail latency, we found that IOQP largely outperformed JASSv2, with a few exceptions, e.g. SPLADEv2 and unicoil-TILDE. Although it is clear that IOQP is overall better for tail latency, we are still unable to determine if QMX or SIMD BP-128 is more efficient overall.

In previous work by Trotman [39], they found QMX to largely outperform other SIMD codecs — including SIMD BP-128. We found little difference in performance between the two. In later work by Trotman & Lilly [42], they found that Elias Delta SIMD had a large space effectiveness loss on Gov2 compared to QMX, but a gain in efficiency — which we did not find, hence our inclusion of only QMX. Moreover, they report their Elias Gamma SIMD as being more efficient and less space effective than QMX. However, this too no longer appears to be the case. Perhaps these results are due to improvements in hardware, or differences in compilers. As an example, Trotman [39] found the performance of a codec to be platform specific. Another possibility is our use of Faster Graph Bisection; this was not employed in either of their works.

We performed this experiment with the intention of finding the main cause of performance discrepancy between JASSv2 and IOQP. To that end, we have been somewhat successful. By replacing Elias Gamma SIMD VB with either QMX or SIMD BP-128, JASSv2 and IOQP perform similarly.

7 CPU

Up to this point, we have used an Intel i7-9800X (3.80 GHz) CPU. However, Mackenzie et al. [28] used dual Intel Xeon Gold 6144 CPUs. Thus, to investigate the impact of using a desktop-grade CPU instead of a server-grade, we introduce a second machine — equipped with an Intel Xeon W-2195 (2.30 GHz) and 256GB of RAM. To keep things consistent, we use the same OS and compiler versions as before.

Within each search engine, across all codecs, modes, and collections, we found that using the Xeon generally decreased query latency. More than that, we wanted to determine if the performance gap changed. To demonstrate this idea we present, in Figure 1, the difference in median latency between JASSv2 and IOQP on the different CPUs. For both approximate and exhaustive processing,

Table 5: Median and 99th percentile latency (ms) for JASSv2 and IOQP on the Robust04 collection. Percentage improvements over EGVB are given in parentheses. † indicates statistically significant improvement over other scores.

Model	JASSv2						IOQP			
	EG VB		QMX		BP-128		QMX		BP-128	
	P_{50}	P_{99}	P_{50}	P_{99}	P_{50}	P_{99}	P_{50}	P_{99}	P_{50}	P_{99}
Exhaustive										
BM25	0.43	1.19	0.37 (14.1)	0.92 (23.1)	0.41 (5.8)	1.03 (13.8)	**0.36†(18.0)**	0.93 (21.9)	0.38 (12.9)	**0.86†(27.7)**
Approximate										
BM25	0.42	0.60	0.37 (10.6)	0.54 (9.2)	0.38 (7.7)	0.58 (3.5)	**0.35†(15.4)**	**0.45†(25.6)**	0.38 (9.6)	0.48† (19.1)

Table 6: Median and 99th percentile latency (ms) for JASSv2 and IOQP on the Gov2 collection. Percentage improvements over EGVB are given in parentheses. † indicates statistically significant improvement over other scores.

Model	JASSv2						IOQP			
	EG VB		QMX		BP-128		QMX		BP-128	
	P_{50}	P_{99}	P_{50}	P_{99}	P_{50}	P_{99}	P_{50}	P_{99}	P_{50}	P_{99}
Exhaustive										
BM25	11.8	52.7	9.8 (17.3)	46.4 (11.8)	**9.6†(19.2)**	40.2†(23.6)	11.5 (3.0)	51.5 (2.3)	11.6 (2.2)	**40.1†(23.8)**
Approximate										
BM25	10.1	25.8	8.5 (16.1)	23.6 (8.5)	**8.3†(18.4)**	19.9 (22.8)	10.5 (−3.6)	14.0 (45.8)	10.2 (−1.3)	**14.6†(43.4)**

Table 7: Median and 99th percentile latency (ms) for JASSv2 and IOQP on the MSMARCO passage collection. Percentage improvement over EGVB given in parentheses. † indicates statistically significant improvement other over scores.

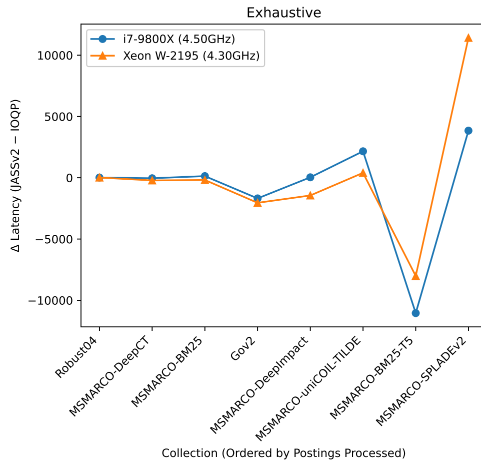
Model	JASSv2						IOQP			
	EG VB		QMX		BP-128		QMX		BP-128	
	P_{50}	P_{99}	P_{50}	P_{99}	P_{50}	P_{99}	P_{50}	P_{99}	P_{50}	P_{99}
Exhaustive										
BM25	6.7	28.2	5.6 (16.5)	22.5 (20.3)	5.6 (15.8)	22.4 (20.4)	**5.4 (18.6)**	22.0 (22.0)	5.5 (18.0)	**20.6†(26.8)**
BM25-T5	18.7	481.4	16.6 (11.3)	481.0 (0.1)	**14.6†(22.0)**	505.0 (−4.9)	27.6 (−47.8)	65.6 (86.4)	15.8 (15.5)	**44.4†(90.8)**
DeepCT	2.9	9.1	**2.5 (13.4)**	7.6 (16.5)	2.6 (9.2)	7.8 (14.3)	**2.5 (12.0)**	**7.1†(21.4)**	2.8 (2.3)	8.1 (10.5)
DeepImpact	25.0	63.7	19.8 (21.0)	51.5 (19.1)	19.2 (23.2)	50.1 (21.3)	19.7 (21.1)	**48.7 (23.5)**	**18.4 (26.6)**	49.3 (22.6)
uniCOIL-TILDE	48.7	197.3	41.2 (15.3)	168.1 (14.8)	41.7 (14.3)	169.8 (13.9)	39.1 (19.7)	162.7 (17.6)	**37.5 (22.9)**	**151.1 (23.4)**
SPLADEv2	227.9	443.5	188.1 (17.5)	376.3 (15.2)	**181.7 (20.3)**	**354.6 (20.1)**	184.2 (19.2)	376.1 (15.2)	182.8 (19.8)	359.0 (19.1)
Approximate										
BM25	6.4	8.2	5.4 (16.0)	7.0 (14.3)	**5.3 (16.7)**	6.9 (15.6)	**5.3 (16.6)**	**6.6†(18.9)**	5.6 (13.4)	7.0 (13.7)
BM25-T5	4.9	18.1	**3.9 (19.1)**	17.2 (4.9)	4.0 (18.2)	17.7 (2.1)	4.0 (17.8)	**6.0†(66.7)**	4.0 (17.2)	6.9 (61.6)
DeepCT	2.9	7.8	**2.6 (11.9)**	6.7 (13.8)	**2.6 (10.1)**	6.7 (14.2)	**2.6 (10.1)**	**6.4†(18.0)**	2.8 (4.6)	6.8 (12.9)
DeepImpact	6.5	8.3	5.5 (14.3)	7.7 (7.3)	5.3 (18.0)	7.2 (14.2)	**5.2 (18.8)**	**6.7†(19.6)**	5.4 (16.9)	7.1 (14.9)
uniCOIL-TILDE	7.3	8.8	6.2 (14.9)	**7.6 (13.5)**	6.3 (14.4)	7.7 (12.9)	**5.9 (19.2)**	**7.6 (13.7)**	6.4 (12.4)	8.3 (6.0)
SPLADEv2	7.7	9.2	6.3 (18.5)	7.7 (16.0)	6.6 (14.3)	8.3 (10.0)	**6.1 (21.5)**	**7.6 (17.4)**	6.5 (15.6)	8.2 (11.2)

we see that JASSv2 is largely outperforming IOQP on the Xeon. We acknowledge that the size of the difference between JASSv2 and IOQP is relative to the CPU frequency, and that we have not normalised by this.

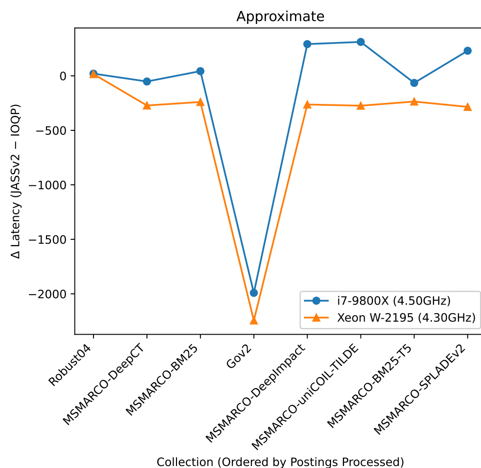
To determine if there was any statistical significance to our results, we performed two-way ANOVA between JASSv2 and IOQP. We used QMX for both engines (the better codec for IOQP), and separately investigated each collection across both exhaustive and approximate processing. Our aim was to determine if the search engine and CPU affected query latency. For exhaustive and approximate processing, we found that, except for exhaustive SPLADEv2, there was an interaction between search engine and CPU. That is to say, query latency is affected by hardware — but the effects are not equal across search engines. Although we found improved performance on the Xeon overall, IOQP was less affected than JASSv2. Indeed, a t-test between JASSv2 and IOQP showed that on collections like DeepCT and DeepImpact, where IOQP had previously significantly outperformed JASSv2 for tail latency, we now find

JASSv2 outperforms. Overall, we found JASSv2 was faster than IOQP on the Xeon.

Our findings highlight the importance of reporting hardware specifications. On the desktop-grade i7 IOQP was the better choice, but for the server-grade Xeon it was JASSv2. Anecdotally, we point out the opposite is true for indexing from a CIFF; IOQP needs server-level resources (i.e. a considerable amount of RAM) while JASSv2 can index on a desktop.



(a) Exhaustive



(b) Approximate

Figure 1: Actual difference in median latency (μs) between JASSv2 and IOQP using QMX. Negative values favour JASSv2. Collections have been ordered by median number of postings processed.

8 Limitations & Pitfalls

In the interest of reproducibility, the driving force behind this work, we detail some difficulties we encountered. Our first pitfall occurred early in the project; we could not get the consistent search results we needed for fair efficiency comparisons. This led to us developing our ciffTools, as detailed in Section 3. We also mention the importance of query standardisation. For Anserini, it is crucial to use the DumpAnalyzedQueries tool to ensure query terms are processed in the same way as the index. Otherwise, there is a mismatch between the query terms and the index terms. Finally, we note that JASSv2 counts documents from 1, but IOQP starts from 0. Though this seems unremarkable, it caused some issues when implementing SIMD BP-128 into JASSv2.

We also address some of the limitations regarding our findings. We did not do multi-threaded throughput tests, instead focusing on compression codecs. But, there are other aspects of the engines that could be influencing the result, such as accumulator management strategies. We could have extended our experiments to collections such as the ClueWeb; we leave this to future work. Although we did not use the same versions of JASSv2 and IOQP as Mackenzie et al. [28], we believe that the minimal changes made since the release of said paper do not invalidate our findings.

9 Insights & Conclusions

In this work, we performed efficiency comparisons between JASSv2 and IOQP to reproduce work by Mackenzie et al. [28]. We also experimented with the compression codecs used by these systems, namely Elias Gamma SIMD VB, QMX, and SIMD BP-128 — thus replicating work by Trotman [39] and Trotman & Lilly [42]. Finally, we concluded a brief investigation into the effect of the CPU. This has led to us gaining several insights into Score-at-a-Time retrieval:

- Different implementations often interpret parameters like ρ differently. For a more like-to-like comparison, we kept the treatment of ρ consistent between the two systems. Similarly, we introduced our ciffTools.
- Compression codec has a significant effect on throughput.
- Faster Graph Bisection likely has an effect on the space effectiveness and efficiency of compression codecs.
- Query latency is affected by CPU architecture differently in each implementation, indicating that it is dependent on more than just CPU frequency. We believe this may be due to differences in accumulator management.

Overall, we found that JASSv2 typically had faster median latency than IOQP with a server-grade CPU, but they were evenly matched on a desktop-grade. For tail latency, IOQP was faster regardless. We found that Elias Gamma SIMD VB was more space effective, but less efficient than the other codecs. Although QMX and SIMD BP-128 were similar across the datasets, we discovered that IOQP particularly benefitted from QMX. Thus, we recommend swapping both systems to QMX. We also suggest that IOQP adopt JASSv2 termination logic, as we found it decreased latency.

We provide QMX in Rust and SIMD BP-128/StreamVByte in C++ through zero-overhead gateways between Rust and C++, the code for which has been made publicly available[6]. Further, in the interest of enabling ease of reproducibility, we provide our open-source ciffTools to facilitate working with CIFF files. Currently, this resource provides a user with a means of quantizing a CIFF index, or printing a human-readable dump of a CIFF index. It is our hope

[6]https://github.com/Axiomatic314/qmx_compression
https://github.com/Axiomatic314/bp_compression

that our work encourages further developments in Score-at-a-Time retrieval.

In future work, we will investigate accumulator management strategies. Preliminary experiments suggest this has a considerable effect on latency. In particular, the performance of the accumulator management may be CPU-sensitive — possibly due to different vectorization or cache sizes. Alongside this, as JASSv2 has been demonstrated to benefit from 8-bit accumulators [32], we will consider both 8-bit and 16-bit accumulators. Finally, it is evident that JASSv2 would benefit from an investigation into its tail latency.

Acknowledgements

The authors acknowledge the peoples of the Woi Wurrung and Boon Wurrung language groups of the eastern Kulin Nation on whose unceded lands ACM SIGIR 2026 was hosted. We pay our respects to their Elders past and present, and extend that respect to all Aboriginal and Torres Strait Islander peoples today and their continuing connection to land, sea, sky, and community.

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*. ACM, New Orleans Louisiana USA, 35–42. doi:10.1145/383952.383957

[2] Michael Antonios Kruse Ayoub, Zhan Su, and Qiuchi Li. 2024. A Case Study of Enhancing Sparse Retrieval using LLMs. In *Companion Proceedings of the ACM Web Conference 2024 (WWW '24)*. ACM, New York, NY, USA, 1609–1615. doi:10.1145/3589335.3651945

[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. doi:10.48550/arXiv.1611.09268 arXiv:1611.09268 [cs].

[4] Sebastian Bruch, Siyu Gai, and Amir Ingber. 2023. An Analysis of Fusion Functions for Hybrid Retrieval. *ACM Trans. Inf. Syst.* 42, 1 (Aug. 2023). doi:10.1145/3596512

[5] Sebastian Bruch, Franco Maria Nardini, Amir Ingber, and Edo Liberty. 2023. An Approximate Algorithm for Maximum Inner Product Search over Streaming Sparse Vectors. *ACM Trans. Inf. Syst.* 42, 2 (Nov. 2023), 42:1–42:43. doi:10.1145/3609797

[6] Sebastian Bruch, Franco Maria Nardini, Amir Ingber, and Edo Liberty. 2024. Bridging Dense and Sparse Maximum Inner Product Search. *ACM Trans. Inf. Syst.* 42, 6 (Aug. 2024), 151:1–151:38. doi:10.1145/3665324

[7] Sebastian Bruch, Franco Maria Nardini, Cosimo Rulli, and Rossano Venturini. 2024. Efficient Inverted Indexes for Approximate Retrieval over Learned Sparse Representations. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, Washington DC USA, 152–162. doi:10.1145/3626772.3657769

[8] Stefan Buttcher, Charles L A Clarke, and Ian Soboroff. 2006. The TREC 2006 Terabyte Track. *Proc. TREC* (2006).

[9] Tao Chen, Mingyang Zhang, Jing Lu, Michael Bendersky, and Marc Najork. 2022. Out-of-Domain Semantics to the Rescue! Zero-Shot Hybrid Retrieval Models. In *Advances in Information Retrieval*. Springer, Cham, 95–110. doi:10.1007/978-3-030-99736-6_7 ISSN: 1611-3349.

[10] Charles Clarke, Nick Craswell, and Ian Soboroff. 2004. Overview of the TREC 2004 Terabyte Track. *Proc. TREC* (2004).

[11] Charles L A Clarke, Falk Scholer, and Ian Soboroff. 2005. The TREC 2005 Terabyte Track. *Proc. TREC* (2005).

[12] Zhuyun Dai and Jamie Callan. 2019. Context-Aware Sentence/Passage Term Importance Estimation For First Stage Retrieval. doi:10.48550/arXiv.1910.10687

[13] Zhuyun Dai and Jamie Callan. 2020. Context-Aware Document Term Weighting for Ad-Hoc Search. In *Proceedings of The Web Conference 2020 (WWW '20)*. Association for Computing Machinery, New York, NY, USA, 1897–1907. doi:10.1145/3366423.3380258

[14] Laxman Dhulipala, Igor Kabiljo, Brian Karrer, Giuseppe Ottaviano, Sergey Pupyrev, and Alon Shalita. 2016. Compressing Graphs and Indexes with Recursive Graph Bisection. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. Association for Computing Machinery, New York, NY, USA, 1535–1544. doi:10.1145/2939672.2939862

[15] P. Elias. 1975. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory* 21, 2 (March 1975), 194–203. doi:10.1109/TIT.1975.1055349

[16] Thibault Formal, Carlos Lassance, Benjamin Piwowarski, and Stéphane Clinchant. 2021. SPLADE v2: Sparse Lexical and Expansion Model for Information Retrieval. doi:10.48550/arXiv.2109.10086

[17] Thibault Formal, Carlos Lassance, Benjamin Piwowarski, and Stéphane Clinchant. 2022. From Distillation to Hard Negative Sampling: Making Sparse Neural IR Models More Effective. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '22)*. Association for Computing Machinery, New York, NY, USA, 2353–2359. doi:10.1145/3477495.3531857

[18] Xiangfei Jia, A. Trotman, and Richard O'Keefe. 2010. Efficient Accumulator initialisation. *Proceedings of the 15th Australasian Document Computing Symposium.*

[19] Chris Kamphuis, Arjen P. de Vries, Leonid Boytsov, and Jimmy Lin. 2020. Which BM25 Do You Mean? A Large-Scale Reproducibility Study of Scoring Variants. In *Advances in Information Retrieval*, Joemon M. Jose, Emine Yilmaz, João Magalhães, Pablo Castells, Nicola Ferro, Mário J. Silva, and Flávio Martins (Eds.). Springer International Publishing, Cham, 28–34. doi:10.1007/978-3-030-45442-5_4

[20] 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. doi:10.48550/arXiv.2004.04906

[21] Carlos Lassance and Stéphane Clinchant. 2022. An Efficiency Study for SPLADE Models. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '22)*. Association for Computing Machinery, New York, NY, USA, 2220–2226. doi:10.1145/3477495.3531833

[22] D. Lemire and L. Boytsov. 2015. Decoding billions of integers per second through vectorization. *Software: Practice and Experience* 45, 1 (2015), 1–29. doi:10.1002/spe.2203

[23] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. 2016. SIMD compression and the intersection of sorted integers. *Software: Practice and Experience* 46, 6 (2016), 723–749. doi:10.1002/spe.2326

[24] Daniel Lemire, Nathan Kurz, and Christoph Rupp. 2018. Stream VByte: Faster byte-oriented integer compression. *Inform. Process. Lett.* 130 (Feb. 2018), 1–6. doi:10.1016/j.ipl.2017.09.011

[25] 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. (March 2020). doi:10.48550/arXiv.2003.08276

[26] Jimmy Lin and Andrew Trotman. 2015. Anytime Ranking for Impact-Ordered Indexes. In *Proceedings of the 2015 International Conference on The Theory of Information Retrieval (ICTIR '15)*. Association for Computing Machinery, New York, NY, USA, 301–304. doi:10.1145/2808194.2809477

[27] Jimmy Lin and Andrew Trotman. 2017. The role of index compression in score-at-a-time query evaluation. *Information Retrieval Journal* 20, 3 (June 2017), 199–220. doi:10.1007/s10791-016-9291-5

[28] Joel Mackenzie, Matthias Petri, and Luke Gallagher. 2022. IOQP: A simple Impact-Ordered Query Processor written in Rust. *Proceedings of the 3rd International Conference on Design of Experimental Search and Information Retrieval Systems* (2022), 22–34.

[29] Joel Mackenzie, Matthias Petri, and Alistair Moffat. 2021. Anytime Ranking on Document-Ordered Indexes. *ACM Trans. Inf. Syst.* 40, 1 (Sept. 2021), 13:1–13:32. doi:10.1145/3467890

[30] Joel Mackenzie, Matthias Petri, and Alistair Moffat. 2021. Faster Index Reordering with Bipartite Graph Partitioning. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '21)*. Association for Computing Machinery, New York, NY, USA, 1910–1914. doi:10.1145/3404835.3462991

[31] Joel Mackenzie, Andrew Trotman, and Jimmy Lin. 2021. Wacky Weights in Learned Sparse Representations and the Revenge of Score-at-a-Time Query Evaluation. (Oct. 2021). doi:10.48550/arXiv.2110.11540

[32] Joel Mackenzie, Andrew Trotman, and Jimmy Lin. 2022. Efficient Document-at-a-Time and Score-at-a-Time Query Evaluation for Learned Sparse Representations. *ACM Transactions on Information Systems* (Dec. 2022), 3576922. doi:10.1145/3576922

[33] 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 '21)*. Association for Computing Machinery, New York, NY, USA, 1723–1727. doi:10.1145/3404835.3463030

[34] Antonio Mallia, Joel Mackenzie, Torsten Suel, and Nicola Tonellotto. 2022. Faster Learned Sparse Retrieval with Guided Traversal. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, Madrid Spain, 1901–1905. doi:10.1145/3477495.3531774

[35] Antonio Mallia, Torsten Suel, and Nicola Tonellotto. 2024. Faster Learned Sparse Retrieval with Block-Max Pruning. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '24)*. Association for Computing Machinery, New York, NY, USA, 2411–2415. doi:10.1145/3626772.3657906

[36] Franco Maria Nardini, Cosimo Rulli, and Rossano Venturini. 2024. Efficient Multi-vector Dense Retrieval with Bit Vectors. In *Advances in Information Retrieval*, Nazli Goharian, Nicola Tonellotto, Yulan He, Aldo Lipani, Graham McDonald, Craig Macdonald, and Iadh Ounis (Eds.). Vol. 14609. Springer Nature Switzerland, Cham, 3–17. doi:10.1007/978-3-031-56060-6_1

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

[38] Stephen E Robertson, Steve Walker, Susan Jones, Micheline M Hancock-Beaulieu, Mike Gatford, et al. 1995. Okapi at TREC-3. *Nist Special Publication Sp* 109 (1995), 109.

[39] Andrew Trotman. 2014. Compression, SIMD, and Postings Lists. In *Proceedings of the 19th Australasian Document Computing Symposium (ADCS '14)*. Association for Computing Machinery, New York, NY, USA, 50–57. doi:10.1145/2682862.2682870

[40] Andrew Trotman and Matt Crane. 2019. Micro and Macro Optimization of SaaT Search. *Software: Practice and Experience* 49, 5 (2019), 942–950. doi:10.1002/spe.2683

[41] A. Trotman, Xiangfei Jia, and Matt Crane. 2012. Towards an Efficient and Effective Search Engine. In *SIGIR 2012 Workshop on Open Source Information Retrieval*.

[42] Andrew Trotman and Kat Lilly. 2018. Elias Revisited: Group Elias SIMD Coding. In *Proceedings of the 23rd Australasian Document Computing Symposium (ADCS '18)*. Association for Computing Machinery, New York, NY, USA, 1–8. doi:10.1145/3291992.3292001

[43] Andrew Trotman and Jimmy Lin. 2016. In Vacuo and In Situ Evaluation of SIMD Codecs. In *Proceedings of the 21st Australasian Document Computing Symposium (ADCS '16)*. Association for Computing Machinery, New York, NY, USA, 1–8. doi:10.1145/3015022.3015023

[44] Andrew Trotman, Antti Puurula, and Blake Burgess. 2014. Improvements to BM25 and Language Models Examined. In *Proceedings of the 19th Australasian Document Computing Symposium (ADCS '14)*. Association for Computing Machinery, New York, NY, USA, 58–65. doi:10.1145/2682862.2682863

[45] Ellen M Voorhees. 2004. Overview of the TREC 2004 Robust Retrieval Track. (2004).

[46] Peilin Yang, Hui Fang, and Jimmy Lin. 2018. Anserini: Reproducible Ranking Baselines Using Lucene. *Journal of Data and Information Quality* 10, 4 (Oct. 2018), 16:1–16:20. doi:10.1145/3239571

[47] Haoyu Zhang, Jun Liu, Zhenhua Zhu, Shulin Zeng, Maojia Sheng, Tao Yang, Guohao Dai, and Yu Wang. 2024. Efficient and Effective Retrieval of Dense-Sparse Hybrid Vectors using Graph-based Approximate Nearest Neighbor Search. doi:10.48550/arXiv.2410.20381

[48] Shengyao Zhuang and Guido Zuccon. 2021. Fast Passage Re-ranking with Contextualized Exact Term Matching and Efficient Passage Expansion. doi:10.48550/arXiv.2108.08513