IOQP: A simple Impact-Ordered Query Processor written in Rust

Joel Mackenzie[1], Matthias Petri[2] and Luke Gallagher[3]

[1]*The University of Queensland, Brisbane, Australia*
[2]*Amazon Alexa, Manhattan Beach, California, USA*
[3]*RMIT University, Melbourne, Australia*

Abstract
Impact-ordered index organizations are suited to score-at-a-time query evaluation strategies. A key advantage of score-at-a-time processing is that query latency can be tightly controlled, leading to lower tail latency and less latency variance overall. While score-at-a-time evaluation strategies have been explored in the literature, there is currently only one notable system that promotes impact-ordered indexing and efficient score-at-a-time query processing. In this paper, we propose an alternative implementation of score-at-a-time retrieval over impact-ordered indexes in the Rust programming language. We detail the efficiency-effectiveness characteristics of our implementation through a range of experiments on two test collections. Our results demonstrate the efficiency of our proposed model in terms of both single-threaded latency, and multi-threaded throughput capability. We make our system publicly available to benefit the community and to promote further research in efficient impact-ordered query processing.

Keywords
Impact-Ordered indexes, Score-at-a-Time retrieval, Learned sparse models, Empirical experimentation

1. Introduction

Despite their simplicity, inverted indexes continue to be an important data structure for efficient and scalable retrieval over large document collections. Inverted indexes maintain, for each unique term discovered during indexing, a list of documents which contain the given term (known as a *postings list*), with perhaps some statistical information such as the number of times each term appeared within each document. These postings lists then allow documents to be efficiently matched and ranked given an input query.

While a range of inverted index organizations and retrieval methods have been investigated, score-at-a-time (SAAT) retrieval over impact-ordered indexes remains a rather under-explored alternative to document-at-a-time (DAAT) retrieval over document-ordered indexes [1]. Indeed, SAAT retrieval was first discussed over two decades ago [2], but has since fallen out of favor with DAAT systems becoming the predominate focus in both academic research and industrial search applications [3, 1, 4].

Recently, SₐₐT retrieval has been revisited in the context of so-called *learned sparse models*, which employ neural networks to learn per-document term weights which can then be stored within an inverted index. Due to the somewhat unconventional term and document weightings that arise from these learned models, DₐₐT dynamic pruning algorithms [5, 6] are not able to effectively exploit the term-wise upper-bound information that usually allows scoring operations to be bypassed, making them less efficient than when used in conjunction with traditional statistical bag-of-words models. SₐₐT retrieval, however, has been shown to provide competitive trade-offs between efficiency and effectiveness in the context of learned sparse retrieval, rivaling even the most efficient DₐₐT algorithms [7].

In this work, we describe IOQP, an *impact-ordered query processor* written in the Rust programming language. We report some preliminary experiments to demonstrate the efficiency and effectiveness of IOQP in both single-threaded retrieval and multi-threaded throughput experiments, and with both traditional and learned sparse retrieval models. Our findings show that SₐₐT retrieval implemented in Rust has comparable efficiency to that of a highly optimized C++ implementation. We hope to encourage further research in this interesting area of Information Retrieval (IR) by contributing to a more diverse ecosystem of available resources.

2. Background

2.1. Document Ranking Models

Traditional bag-of-words (BoW) ranking models such as BM25 [8, 9] assume *term independence*, and score documents as a sum of term-document contributions across the terms in a given query. Recently, however, a new class of learned sparse ranking models have been investigated by the IR community [10, 11, 12, 13, 14, 15, 16, 17]. These models are derived by training neural networks (typically transformer-based models such as BERT) in a supervised manner to *learn* term-document contributions. During inference, a learned sparse model generates a prediction for each term-document *impact* which is then stored within a classical inverted index structure. Then, documents can be scored by summing these impacts across query terms.

2.2. Impact-Ordered Indexing and Score-at-a-Time Retrieval

Impact-ordered indexes organize each postings list into a number of *segments*, each of which is representative of a given term-document impact score. These impacts are typically pre-computed during indexing. Since many ranking models produce floating point scores which are difficult to compress, a *quantized integer* representation is stored instead. These integers are typically generated by uniformly quantizing the entire floating point score range into the integers in the range $[0, 2^b - 1]$, with b representing the number of bits required to store each quantity [2, 18]. Within each segment, a list of strictly increasing document identifiers is maintained. Figure 1 demonstrates this simple arrangement.

These impact-ordered postings lists allow for SₐₐT retrieval, where candidate segments can be visited in descending order of their impact score, thereby allowing high scoring documents to be rapidly identified. Currently, JASSv2 [19, 20] is the only open-source SₐₐT retrieval framework available to the community.
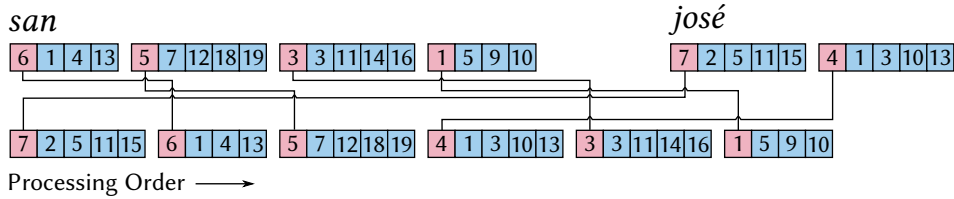
Figure 1: A sketch of an impact-ordered index with two postings lists. Each list consists of a number of segments with an impact score (red) and a sequence of document identifiers (blue). The list for *"san"* has four segments with impacts 6, 5, 3, and 1, containing a total of 14 documents. Similarly, the list for *"josé"* has two segments with impacts 7 and 4, and a total of 8 documents. The lower part of the figure shows how segments are sorted and processed from high-to-low in a "score-at-a-time" order.

3. IOQP

Now, we describe the implementation of IOQP, including the indexing and retrieval components, and describe how IOQP can handle multiple incoming requests over a HTTP endpoint.

3.1. Indexing

Instead of building an indexing pipeline, we implemented IOQP to read pre-built *common index file format* (CIFF) indexes [21]. This allows the complexities of indexing to be outsourced to other systems, and provides better integration with existing Rust tools [22]. Although the CIFF provides pre-built indexes which can be accessed through a standardized protobuf API, these indexes need to be re-written into a suitable format for use within IOQP. In the most simple case, where the postings inside the CIFF are pre-quantized, this involves iterating the CIFF structure and re-organizing the underlying data into the IOQP index format. In the more complex case, where the CIFF structure contains raw term-frequency information, it will need to be scored and quantized by IOQP before it can be indexed; to support this, we implemented a variant of the common BoW BM25 model [9] and a uniform score quantizer [2, 18] which converts floating point scores into quantized impacts in a fixed range $[0, 2^b - 1]$. These indexing processes are implemented with parallel processing capability.

The document identifiers within each segment are delta-coded and compressed with SIMD-enabled bitpacking algorithms (SIMD-BP) [23, 24]. The specific implementation of SIMD-BP can operate on blocks of 128 or 256 integers, depending on the CPU instructions available on the target system; shorter blocks are encoded with StreamVByte [25].

3.2. Query Processing

Score-at-a-time query processing is quite simple. In IOQP, each query is assigned a `Scratch` data structure (the in-memory representation required to execute a single query). This `Scratch` data structure contains: a vector for managing the candidate segments; buffers for decoding the segments; an accumulator table for tracking document scores, and; a min-heap for finding the top-k ranked documents. There are three key steps for processing a query: (1) determine which impact segments to process; (2) process those segments, and; (3) determine the top-k documents from the scores in the accumulator table.
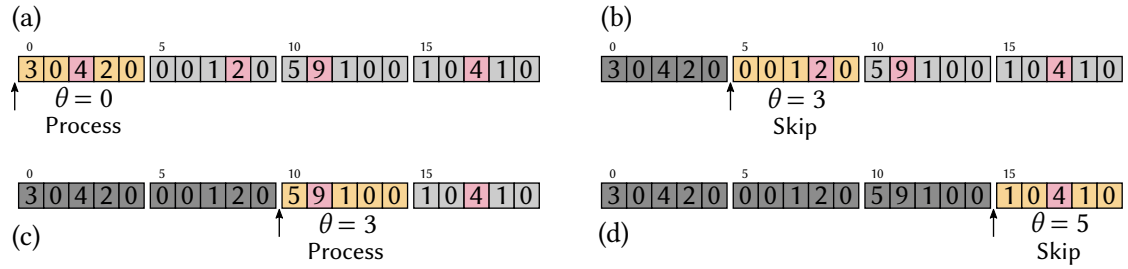
Figure 2: Processing the accumulator array to find the top 2 documents with a chunk size $W = 5$ and with 20 documents in total. Within each chunk, the top score is highlighted in red; yellow chunks are those currently under consideration; and dark gray chunks have already been considered. (a) The first chunk of accumulators is processed to initialize the min-heap (not shown) and threshold θ. (b) The second chunk is skipped since $\theta > 2$, meaning that no document in the second chunk could enter the min-heap. (c) The third chunk is processed since $\theta < 9$; there is at least one document in the third chunk which will enter the min-heap. (d) The fourth chunk is skipped since $\theta > 4$. At this stage, the min-heap would contain the top 2 documents, 11 and 10, with scores 9 and 5 respectively.

Determining which Segments to Process   The first step is to decide which segments should be processed, and the order in which they should be processed. This involves arranging a vector of metadata entries which, for each candidate segment, describe the location of the segment in memory, the impact of the segment, and the number of entries it holds. The vector is arranged such that the segments are visited in descending order of the impacts.

Processing the Segments   The next step is to process the segments. At this stage, the accumulator table may contain data from the previous query, so it needs to be cleared. We rely on the Rust compiler to optimize this process, although other optimizations have been investigated [26, 27]. Then, while fewer than ρ postings have been visited, the next candidate segment is decoded into the buffer; for each document in the segment, the impact score is added to the corresponding accumulator. Also maintained is the maximum score of a document within each chunk of W accumulators; we set $W = 128$ in our experiments, and leave finding the best choice of W to future work. It is also worth noting that in IOQP, term impacts can either be unweighted, or weighted according to the number of times each term appears in the query; the only difference to processing is the value of the impact added to the accumulator table.

Determining the Top Documents   Finally, the top-k documents need to be returned to the caller. Unlike JASSv2, IOQP does not keep track of the k highest scoring documents during query processing, so they need to be found by scanning the accumulator table. Since linearly scanning the entire accumulator table is inefficient, we push the first k accumulators into the min-heap to establish an entry threshold, θ. Then, since we maintain the maximum score of each chunk of W accumulators, we traverse these chunks and only enter and scan those which exceed the current value of θ. Once all chunks have been traversed, the heap contains the top-k scoring documents. Figure 2 demonstrates this novel accumulator skipping strategy.

Early Termination   Similar to the existing JASSv2 system, IOQP supports both exhaustive and approximate processing modes. A parameter, ρ, determines the raw number of postings to be processed, and can be set either as a constant value, or as a proportion of the available postings on a query-by-query basis.[1] IOQP will always process *at least* ρ postings, whereas JASSv2 terminates before the number of postings processed *exceeds* ρ.

3.3. IOQP Server

In production IR environments, systems are provisioned such that multiple incoming queries can be served simultaneously. Within IOQP, we provide a `serve` binary which hosts an index in-memory, and provides a listener bound to an IP address and port. The listener waits for incoming queries, represented as JSON objects, and hands them off to asynchronous processing threads; the results are then returned to the caller as a JSON object. To ensure the index processing server does not become overwhelmed under a high query load, the number of processing cores can be limited during start-up.

4. Experiments

4.1. Experimental Setup

IOQP[2] is written in Rust, and was compiled with `rustc 1.61` using `-O3` optimization as per the default release profile. We use the most recent version of JASSv2[3] as a point of comparison. For fairness, we modified the timing measurement within JASSv2 to ignore the cost of parsing query terms, as this is not part of the latency measurement in IOQP. All collections were indexed using Anserini [3] and converted to CIFF files. Those CIFF files were then reordered with recursive graph bisection [28, 29, 22],[4] which has been shown to accelerate SAAT traversal [30]. For both IOQP and JASSv2, 8-bit quantization was used, and the score accumulators are represented by 16-bit unsigned integers; where weighted queries are used, they are re-normalized to avoid accumulator overflows (since some scores may exceed $2^{16} - 1$). We deploy both IOQP and JASSv2 with two processing modes: *exhaustive* processing involves processing all candidate postings for each query; *approximate* processing uses $\rho = 0.1 \times |D|$, where $|D|$ is the number of documents in the collection [19]. All retrieval runs are computed to depth $k = 1,000$.

Experiments were conducted in-memory on an otherwise idle machine equipped with $2\times$ Intel Xeon Gold 6144 CPUs and 512 GiB of RAM; each CPU socket has access to 256 GiB of RAM, implying a non-uniform memory access (NUMA) architecture.

Collections and Queries   We employ the MS MARCO passage collection [31], which contains around 8.8 million passages. We use a range of traditional and neural augmented retrieval models including BM25 on the original index (BM25) [9], BM25 with a DocT5Query expanded index (BM25-T5) [13], and BM25 scoring over a DeepCT weighted index (DeepCT) [10, 11]. We

---

[1] All postings are processed when the number of postings in a query is less than ρ.
[2] See: github.com/jmmackenzie/ioqp (commit `b7488e`).
[3] See: github.com/andrewtrotman/JASSv2 (commit `5ba7f1`).
[4] See: github.com/mpetri/faster-graph-bisection

also used learned sparse models including DeepImpact [12], uniCOIL [14, 15] with a TILDE [16, 17] expanded index (uniCOIL-TILDE), and SPLADEv2 [32]. All models used the same parameters as those described in prior work [7, 33]. All experiments on MS MARCO used the dev queries, and measure effectiveness with RR@10.

We also experimented with Gov2, a 25 million document crawl of .gov domains, using the title queries from TREC terabyte track topics 701—850 [34, 35, 36], and the 60,000 TREC million query track (MQT) queries [37, 38, 39]. Effectiveness is measured with AP, the official metric.

4.2. Indexing

Our first experiment briefly reports on the indexing time and index size of the IOQP indexes. In our experiments, the MS MARCO passage indexes took between 13 and 90 seconds to generate, depending on the specific scoring model (as the number of postings lists varies widely). The larger Gov2 index took just under 7 minutes to generate, which includes scoring and quantizing the index. IOQP made use of all 32 threads for indexing; JASSv2 indexing (from the same CIFF starting point) was much slower as it is not currently multi-threaded. On the other hand, the resulting IOQP indexes are between 10 to 25% larger than the equivalent JASSv2 indexes; this represents an overhead of around 3 GiB on the larger Gov2 collection (10 vs 13 GiB). We aim to optimize the index space consumption in future work.

4.3. Efficiency and Effectiveness

Our next series of experiments validates the performance and effectiveness of IOQP, as compared to the JASSv2 system. In particular, we partially replicate the recent work of Mackenzie et al. [7] which examines the trade-offs arising with different learned sparse retrieval models on the MS MARCO passage collection.

Table 1 shows the efficiency in terms of mean, median (P_{50}), and 99 th percentile (P_{99}) latency, as well as the effectiveness in terms of RR@10, across different models and configurations on the MS MARCO dev queries. Considering exhaustive processing, both JASSv2 and IOQP follow similar trends, with more effective models generally requiring more processing time. One interesting exception is the DeepCT model; since DeepCT implicitly prunes low impact terms (by setting their weight to zero during indexing), the postings lists are shorter on average, resulting in faster retrieval [40]. In general, IOQP outperforms JASSv2, especially in terms of high percentile tail latency.

Turning to the approximate results, we again see IOQP outperforming JASSv2 for most of the metrics. One clear exception is the high P_{99} latency observed for IOQP on the BM25-T5 index. Failure analysis revealed that this is due to the difference in termination decision logic between JASSv2 and IOQP; since IOQP terminates processing once *at least* ρ postings have been considered, it is vulnerable to corner cases which can greatly exceed the expected processing budget. In this specific example, IOQP is processing a single segment of around 8 million postings corresponding to the term *"what,"* whereas JASSv2 simply decides to terminate before processing this block, leading to a large difference in tail latency. It may be worth adapting the JASSv2 termination logic into IOQP to avoid this issue in the future.

Table 1

Mean, median and 99 th percentile latency (ms) and reciprocal rank scores of JASSv2 (left) and IOQP (right) using different ranking models on the MS MARCO collection (dev queries) with exhaustive processing (top) and approximate processing (bottom).

Model	JASSv2				IOQP			
	Mean	P_{50}	P_{99}	RR	Mean	P_{50}	P_{99}	RR
Exhaustive								
BM25	8.2	6.5	30.5	0.186	5.8	4.9	17.5	0.188
BM25-T5	38.9	27.8	381.6	0.277	16.4	17.5	41.2	0.265
DeepCT	2.9	2.7	8.1	0.243	2.7	2.5	5.9	0.243
DeepImpact	23.1	24.4	60.7	0.326	16.0	16.7	41.1	0.326
uniCOIL-TILDE	51.9	43.6	171.4	0.350	36.9	31.2	119.0	0.350
SPLADEv2	217.7	216.0	400.5	0.369	153.7	152.2	284.5	0.368
Approximate								
BM25	5.7	6.6	7.8	0.185	4.4	4.8	6.0	0.186
BM25-T5	5.0	6.0	7.8	0.273	8.0	4.9	20.8	0.264
DeepCT	2.9	2.6	7.6	0.242	2.8	2.6	5.9	0.242
DeepImpact	6.1	6.5	7.8	0.318	5.0	5.4	6.3	0.318
uniCOIL-TILDE	7.1	7.2	8.0	0.335	5.9	6.0	6.8	0.336
SPLADEv2	7.7	7.6	8.8	0.319	5.9	5.9	7.3	0.318

Table 2

Mean, median and 99 th percentile latency (ms) and average precision scores of JASSv2 (left) and IOQP (right) using BM25 on the Gov2 collection (topics 701—850) for both exhaustive processing (top) and approximate processing (bottom).

Model	JASSv2				IOQP			
	Mean	P_{50}	P_{99}	AP	Mean	P_{50}	P_{99}	AP
Exhaustive								
BM25	20.2	15.6	69.1	0.306	16.7	13.9	43.1	0.306
Approximate								
BM25	12.3	13.8	29.3	0.301	12.6	13.9	18.6	0.301

Table 2 compares the efficiency and effectiveness of both JASSv2 and IOQP for BM25 retrieval on the larger Gov2 document collection. Once again, IOQP outperforms JASSv2 on the exhaustive queries, while achieving the same effectiveness. For approximate processing, the systems are much closer; JASSv2 is slightly faster at the mean and median latency, but IOQP has a lower tail latency.

4.4. Throughput

Our final experiment aims to measure the performance of IOQP under a more realistic workload, where multiple incoming queries must be processed as rapidly as possible. We use a load
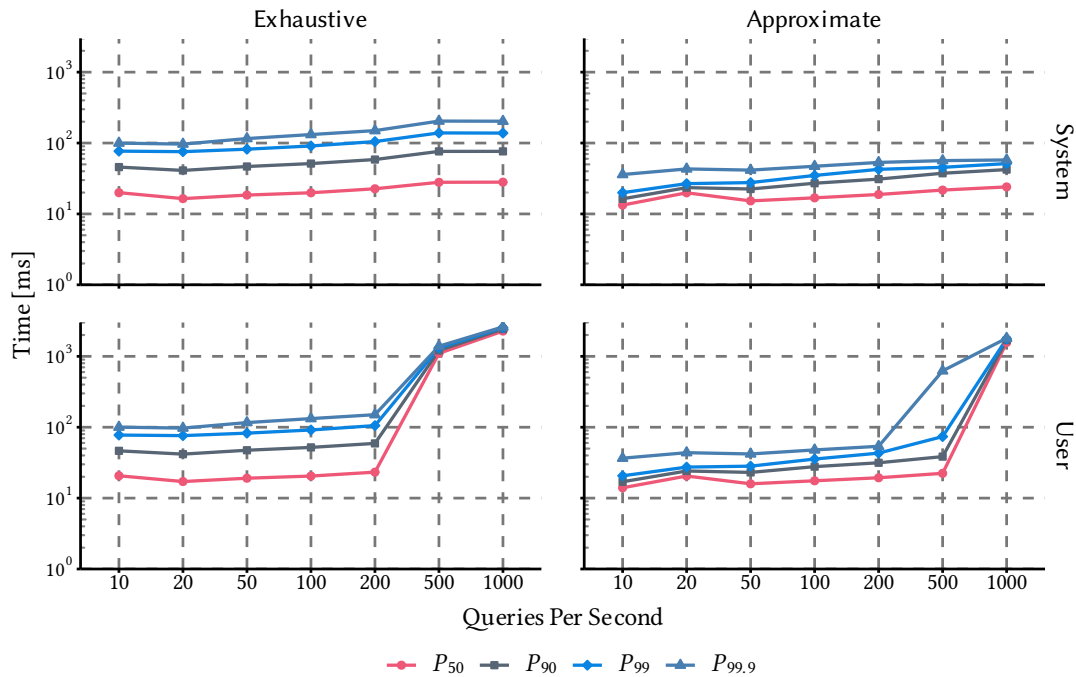
Figure 3: Latency percentiles for 300 seconds worth of incoming queries at different rates, faceted by system and user time measurements, and by exhaustive and approximate processing modes. This experiment uses the Gov2 collection with the entire million query track log shuffled randomly. The underlying server uses 16 physical cores for query processing.

generator to send queries from the MQT log to the IOQP server, at a uniform rate, via HTTP requests. The IOQP server uses 16 cores for processing,[5] and queries are processed in a *first-in, first-out* order. In our experiments, latency is measured over a 300 second window, and the latency measurements over the first 1,000 requests are discarded as a "warm-up" phase.

Figure 3 shows the query response time for both exhaustive and approximate processing of incoming queries at different uniform rates. The top facet shows the system-side response time, which represents the total wall-clock time each query spends on the CPU. The bottom facet shows the user-side response time, which represents the wall-clock time elapsed from the moment the query is submitted, to the moment the results are returned. While the tolerable latency is dependent on a number of factors, the most important aspect is that it is measured on the user-side, since user-side latency is correlated with user experience [41, 42, 43].

Based on our experimental configuration, exhaustive processing can cope with between 200 and 500 queries per second while maintaining acceptable user-side latency. Similarly, approximate processing remains acceptable at 500 queries per second, assuming the 99.9 th percentile tail latency is not a limitation. At higher loads, as the queue of queries grows, queuing times increase drastically, adding to user-side latency. Nevertheless, system latency is quite stable as load increases.

---

[5]The load generator runs with one thread on the same physical system, sending requests to `localhost`.

5. Perplexing Latency Spikes

Our final discussion focuses on a somewhat perplexing "bug" we ran into during the development of IOQP.[6] During prototyping, we noticed that while median latency was stable, the extreme tail latency ($P_{99.9}$ and above) was uncharacteristically high (up to $6\times$ the P_{50}). While this is plausible for exhaustive processing, where query length or list density can affect latency, we observed the same behavior with approximate processing (under strict processing budgets).

To debug this problem, we added a non-blocking "tracing" thread to collect and output runtime diagnostic information. Interestingly, the introduction of this diagnostic thread modified program behavior causing the latency spikes to *disappear entirely*,[7] returning performance to a nominal latency profile. This unsatisfying result led us toward a set of different strategies to diagnose the problem. Turning back to the original binary (without the diagnostics), we applied some simple per-function instrumentation. The purpose of this instrumentation was to determine which specific function call in the IOQP codebase was the root cause of the latency spikes. Surprisingly, we found that no specific function was to blame; the latency spikes were distributed across *all function calls* proportional to the program execution time associated with each given function. At this stage, we concluded the effect was a lower level systems issue external to IOQP itself, and caused by interactions between IOQP and the operating system (OS) or hardware.

After a number of experiments and hypotheses concerning CPU frequency throttling, thermal issues, and migrations, we narrowed down the cause of the latency spikes to be a *memory issue*. The first clue was that enabling transparent huge pages (THP), which allows the operating system to allocate much larger memory pages than the default page size (2 MiB vs 4 KiB), changed the pattern and frequency of the latency spikes. Secondly, we could only reliably reproduce the erroneous behavior on certain systems (our second encounter with non-deterministic diagnostic outcomes while attempting to find the source of the problem). This strongly indicated that system configuration, in the hardware or OS, was the likely cause of our latency troubles.

Detailed profiling experiments with tools such as `perf`, `ftrace`, and `flamegraph` [45] eventually led us to the root cause of the latency spikes: a large proportion of time was being spent by the OS kernel inside a function called `task_numa_work`. This specific function is responsible for balancing memory pages in NUMA systems, which can invoke memory page faults and migrations, causing stalls during processing. In our case, translation lookaside buffer (TLB) flushes were happening routinely when NUMA was enabled.

Figure 4 demonstrates this effect in isolation, where the same query is processed repeatedly and the per-query latency is measured with different NUMA configurations. Interestingly, disabling NUMA completely resolved these spikes. Hence, all of the experimentation in Section 4 was conducted with automatic NUMA balancing *deactivated*. Nevertheless, the lesson here is that correctly benchmarking software performance is difficult, and care must be taken to mitigate external effects which can bias measurement [46]. NUMA is one such source of bias that must be explicitly documented in experimental configurations [47], and/or considered during the design of memory access operations [48, 49].

---

[6]Interestingly, correspondence with Santhanam et al. [44] revealed that they ran into similar issues during the development of their PLAID system.

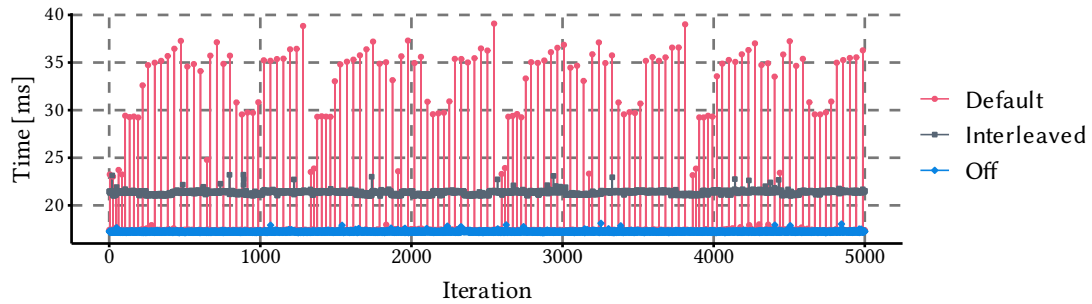[7]This is known as a "Heisenbug," see: en.wikipedia.org/wiki/Heisenbug.

Figure 4: Latency on a per-query basis for 5000 iterations of a randomly selected query (*"civil air patrol"*) on Gov2 with exhaustive processing and different NUMA configurations.

6. Conclusion

In this work, we have proposed IOQP, a new impact-ordered query processing system which is written in Rust. We briefly outlined the implementation of IOQP, including how IOQP indexes collections and processes queries. Index construction utilized parallelism and the CIFF interchange format, while the query processing context is one that is in-memory and on-demand. We compared IOQP to the only current impact-ordered querying system, JASSv2, in the context of both traditional ranking models, and a set of recent learned sparse rankers, demonstrating IOQP's competitive performance. We also experimented with a more realistic high-volume querying scenario, allowing us to characterize the performance limits of IOQP on our experimental hardware. Finally, we shared a cautionary tale on the difficulties of accurately benchmarking high performance software.

In future work, we plan to optimize the space consumption of IOQP via a more compact vocabulary representation compared to the simple hash table which is currently used. We are also interested in conducting a deeper analysis of our novel accumulator table strategy, and comparing it to alternatives used by JASSv2 and in the literature. Finally, we plan on comparing IOQP to other systems in our multi-threaded throughput benchmark to gain a better understanding of the relative performance of different indexing and query processing strategies under a more realistic experimental setting.

Software   The source code, and scripts to replicate our experiments, can be found on the IOQP repository: github.com/jmmackenzie/ioqp.

Acknowledgments

References

[1] M. Crane, J. S. Culpepper, J. Lin, J. Mackenzie, A. Trotman, A comparison of document-at-a-time and score-at-a-time query evaluation, in: Proc. WSDM, 2017, pp. 201–210.

[2] V. N. Anh, O. de Kretser, A. Moffat, Vector-space ranking with effective early termination, in: Proc. SIGIR, 2001, pp. 35–42.

[3] P. Yang, H. Fang, J. Lin, Anserini: Reproducible ranking baselines using lucene, J. Data and Information Quality 10 (2018).

[4] K. M. Risvik, T. Chilimbi, H. Tan, K. Kalyanaraman, C. Anderson, Maguro, a system for indexing and searching over very large text collections, in: Proc. WSDM, 2013, pp. 727–736.

[5] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, J. Zien, Efficient query evaluation using a two-level retrieval process, in: Proc. CIKM, 2003, pp. 426–434.

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

[7] J. Mackenzie, A. Trotman, J. Lin, Wacky weights in learned sparse representations and the revenge of score-at-a-time query evaluation, arXiv:2110.11540 (2021).

[8] S. E. Robertson, S. Walker, Some simple effective approximations to the 2-Poisson model for probabilistic weighted retrieval, in: Proc. SIGIR, 1994, pp. 232–241.

[9] S. Robertson, H. Zaragoza, The probabilistic relevance framework: BM25 and beyond, Found. Trends Inf. Ret. 3 (2009) 333–389.

[10] Z. Dai, J. Callan, Context-aware sentence/passage term importance estimation for first stage retrieval, arXiv:1910.10687 (2019).

[11] Z. Dai, J. Callan, Context-aware document term weighting for ad-hoc search, in: Proc. WWW, 2020, pp. 1897–1907.

[12] A. Mallia, O. Khattab, T. Suel, N. Tonellotto, Learning passage impacts for inverted indexes, in: Proc. SIGIR, 2021, pp. 1723–1727.

[13] R. Nogueira, J. Lin, From doc2query to docTTTTTquery, Technical Report, 2019. URL: https://cs.uwaterloo.ca/~jimmylin/publications/Nogueira_Lin_2019_docTTTTTquery-latest.pdf.

[14] L. Gao, Z. Dai, J. Callan, COIL: Revisit exact lexical match in information retrieval with contextualized inverted list, in: Proc. NAACL, 2021, pp. 3030–3042.

[15] J. Lin, X. Ma, A few brief notes on DeepImpact, COIL, and a conceptual framework for information retrieval techniques, arXiv:2106.14807 (2021).

[16] S. Zhuang, G. Zuccon, TILDE: Term independent likelihood moDEl for passage re-ranking, in: Proc. SIGIR, 2021, pp. 1483–1492.

[17] S. Zhuang, G. Zuccon, Fast passage re-ranking with contextualized exact term matching and efficient passage expansion, arXiv:2108.08513 (2021).

[18] M. Crane, A. Trotman, R. O'Keefe, Maintaining discriminatory power in quantized indexes, in: Proc. CIKM, 2013, pp. 1221–1224.

[19] J. Lin, A. Trotman, Anytime ranking for impact-ordered indexes, in: Proc. ICTIR, 2015, pp. 301–304.

[20] A. Trotman, M. Crane, Micro- and macro-optimizations of SaaT search, Soft. Prac. & Exp. 49 (2019) 942–950.

[21] J. Lin, J. Mackenzie, C. Kamphuis, C. Macdonald, A. Mallia, M. Siedlaczek, A. Trotman,

A. de Vries, Supporting interoperability between open-source search engines with the common index file format, in: Proc. SIGIR, 2020, pp. 2149–2152.

[22] J. Mackenzie, M. Petri, A. Moffat, Faster index reordering with bipartite graph partitioning, in: Proc. SIGIR, 2021, pp. 1910–1914.

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

[24] D. Lemire, L. Boytsov, N. Kurz, SIMD compression and the intersection of sorted integers, Soft. Prac. & Exp. 46 (2016) 723–749.

[25] D. Lemire, N. K. C. Rupp, Stream VByte: Faster byte-oriented integer compression, Inf. Proc. Lett. 130 (2018) 1–6.

[26] X.-F. Jia, A. Trotman, R. O'Keefe, Efficient accumulator initialization, in: Proc. ADCS, 2010, pp. 44–51.

[27] J. Mackenzie, F. Scholer, J. S. Culpepper, Early termination heuristics for score-at-a-time index traversal, in: Proc. ADCS, 2017, pp. 8.1–8.8.

[28] L. Dhulipala, I. Kabiljo, B. Karrer, G. Ottaviano, S. Pupyrev, A. Shalita, Compressing graphs and indexes with recursive graph bisection, in: Proc. KDD, 2016, pp. 1535–1544.

[29] J. Mackenzie, A. Mallia, M. Petri, J. S. Culpepper, T. Suel, Compressing inverted indexes with recursive graph bisection: A reproducibility study, in: Proc. ECIR, 2019, pp. 339–352.

[30] J. Mackenzie, M. Petri, A. Moffat, Anytime ranking on document-ordered indexes, ACM Trans. Inf. Syst. 40 (2022) 13:1–13:32.

[31] P. Bajaj, D. Campos, N. Craswell, L. Deng, J. Gao, X. Liu, R. Majumder, A. McNamara, B. Mitra, T. Nguyen, M. Rosenberg, X. Song, A. Stoica, S. Tiwary, T. Wang, MS MARCO: A Human Generated MAchine Reading COmprehension Dataset, arXiv:1611.09268v3 (2018).

[32] T. Formal, C. Lassance, B. Piwowarski, S. Clinchant, SPLADE v2: Sparse lexical and expansion model for information retrieval, arXiv:2109.10086 (2021).

[33] A. Trotman, J. Mackenzie, P. Parameswaran, J. Lin, A common framework for exploring document-at-a-time and score-at-a-time retrieval methods, in: Proc. SIGIR, 2022, pp. 3229–3234.

[34] C. L. A. Clarke, N. Craswell, I. Soboroff, Overview of the TREC 2004 terabyte track, in: Proc. TREC, 2004.

[35] C. L. A. Clarke, F. Scholer, I. Soboroff, Overview of the TREC 2005 terabyte track, in: Proc. TREC, 2005.

[36] S. Büttcher, C. L. A. Clarke, I. Soboroff, Overview of the TREC 2006 terabyte track, in: Proc. TREC, 2006.

[37] J. Allan, B. Carterette, J. A. Aslam, V. Pavlu, B. Dachev, E. Kanoulas, Million query track 2007 overview, in: Proc. TREC, 2007.

[38] J. Allan, J. A. Aslam, B. Carterette, V. Pavlu, E. Kanoulas, Million query track 2008 overview, in: Proc. TREC, 2008.

[39] B. Carterette, V. Pavlu, H. Fang, E. Kanoulas, Million query track 2009 overview, in: Proc. TREC, 2009.

[40] J. Mackenzie, Z. Dai, L. Gallagher, J. Callan, Efficiency implications of term weighting for passage retrieval, in: Proc. SIGIR, 2020, pp. 1821–1824.

[41] X. Bai, I. Arapakis, B. B. Cambazoglu, A. Freire, Understanding and leveraging the impact of response latency on user behaviour in web search, ACM Trans. Inf. Syst. 36 (2017) 1–42.

[42] S.-W. Hwang, S. Kim, Y. He, S. Elnikety, S. Choi, Prediction and predictability for search query acceleration, ACM Trans. Web 10 (2016) 19.1–19.28.

[43] J. Dean, L. A. Barroso, The tail at scale, Commun. ACM 56 (2013) 74–80.

[44] K. Santhanam, O. Khattab, C. Potts, M. Zaharia, PLAID: An efficient engine for late interaction retrieval, arXiv:2205.09707 (2022).

[45] B. Gregg, The flame graph, Commun. ACM 59 (2016) 48–57.

[46] W. Webber, A. Moffat, In search of reliable retrieval experiments, in: Proc. ADCS, 2005, pp. 26–33.

[47] S. Gog, M. Petri, Optimized succinct data structures for massive data, Soft. Prac. & Exp. 44 (2014) 1287–1314.

[48] D. Hawking, B. Billerbeck, Efficient in-memory, list-based text inversion, in: Proc. ADCS, 2017, pp. 1.5–1.8.

[49] Y. Li, I. Pandis, R. Müller, V. Raman, G. M. Lohman, NUMA-aware algorithms: the case of data shuffling, in: Proc. CIDR, 2013.