Fast, Compact, Immediate-Access Indexing for Learned Sparse Retrieval Systems

Billy Rule[0009−0002−7118−3599] and Joel Mackenzie[0000−0001−7992−4633]

The University of Queensland, St Lucia, Australia

Abstract. Learned sparse retrieval (LSR) is an emerging paradigm that uses pretrained language models to assign learned weights to the terms of a document, enabling practitioners to deploy next-generation rankers within their existing lexical retrieval pipelines. Although LSR systems have been found to provide strong increases in effectiveness over traditional statistical approaches, this boost comes at the cost of both indexing and retrieval efficiency. In this work, we explore the application of LSR to a practical online setting where new documents must be indexed and searchable as soon as they arrive. In particular, we create a clean-room re-implementation of the current state-of-the-art linked block dynamic indexing approach, and propose a set of important augmentations that enable efficient online indexing and query processing to generalize to the learned sparse regime. Our results over two traditional and two LSR models, and a multitude of experimental settings, demonstrate the practicality of our approach, allowing new documents to be queried at a reasonable latency while maintaining fast insertion ability.

Keywords: Dynamic Indexing · Learned Sparse Retrieval

1 Introduction

The rise of pretrained language models has introduced a swathe of new techniques to the field of Information Retrieval (IR), allowing for more accurate – yet more expensive – retrieval. One such example is *learned sparse retrieval* (LSR), a family of techniques that are used to augment documents with *learned* term weights, allowing traditional IR search structures and toolkits to be repurposed without significant modification.

Although LSR systems have been extensively tested in the *static* context, where the index is built once, and is then fixed for the remainder of experimentation, there has been little consideration on the *dynamic* context, where new documents must be indexed upon arrival, while still supporting querying. As such, while an efficient static index must only consider query latency and index size, a dynamic index must balance said factors with insertion latency as well, and while recent work has explored this problem in detail with traditional ranking approaches [12, 26], there is yet to be any experimentation in the context of LSR systems. Hence, we are motivated to explore the intersection of dynamic

indexing and LSR in an effort to improve the practicality of the current, static approaches.

Contributions. In this work, we reproduce and extend the current state-of-the-art *linked block* dynamic indexing system [26] to investigate its application to learned sparse retrieval models. Our contributions are as follows:

- We revisit the problem of dynamic indexing in the context of learned sparse retrieval systems;
- We reproduce the state-of-the-art *linked block* index, then propose a set of simple, yet effective, modifications that improve its efficiency when paired with LSR models; and
- We present a comprehensive experimental analysis of index size, indexing throughput, and query latency on both traditional and learned sparse models in the dynamic indexing context.

Our findings demonstrate that the linked block approach remains competitive to strong, static indexing baselines in terms of both index size and query latency, especially for smaller index sizes, while supporting microsecond-level insertion of new documents. Furthermore, we show the importance of supporting dynamic pruning querying algorithms, especially when LSR models are being considered. These findings support the development and application of LSR models in practical, real-world scenarios.

2 Background

Text Indexing. An *inverted index* stores a *posting list* for each term. Each postings list stores information about which documents contain the given term in the document collection, and a payload – such as a term frequency, or an impact score – to assist with ranking. A *vocabulary* maps each term to its respective posting list [36, 41].

An important distinction is between *static* and *dynamic* indexes. A static index is constructed over a fixed collection of documents. It cannot be modified (i.e., adding or removing documents) after construction, and construction is typically assumed to be an *offline process*, allowing significant efficiency optimizations to be made by reorganizing the index. In contrast, a dynamic index must be able to support intake of new documents [18, 41], have said documents be immediately searchable [7, 26], and it may also support modification or deletion of documents [12]. While an efficient index of both types must support low-latency queries and maintain a small space footprint, a dynamic index has the additional requirement of low document insertion latency. Since insertion and querying can occur at any time, it is important that both are sufficiently fast, resulting in a tension between the efficiency of insertion, querying, and overall space consumption [8, 26].

Recently, two novel approaches have been suggested for immediate access indexing. The *apoptosic* index [12] operates on a fixed-sized slab of memory, allocating postings into a circular buffer with pointers stored from each posting

to the next posting; deletion occurs naturally as postings are overwritten in the buffer. The *linked block* index [26] stores posting lists as chains of fixed-sized blocks of data in a fixed-sized slab of memory; it supports rapid insertion, but no deletion. We base our investigation on the latter structure for two reasons. Firstly, is has almost all of the required elements to support LSR methods out-of-the-box; and secondly, its structure can be readily converted to a fixed, static index, an important consideration given the operational context outlined shortly.

Ranking and Retrieval. To retrieve the top-k ranked documents in response to a user query, documents are commonly ranked according to the sum of the term-document weights (or impacts). Traditional models like BM25 store statistics in the inverted index such that these impacts can be computed on-demand [14, 34]; however, impacts can be pre-computed *offline*, quantized into integers, and stored inside the inverted index to accelerate query processing [1, 10].

Learned sparse retrieval models also make use of this additive scoring approach. However, instead of using a statistical model to compute impacts, LSR models rely on pretrained language models to assigns weights to each term based on their estimated importance to the document, which are then stored in the inverted index. LSR models exhibit strong improvements in effectiveness when compared to traditional methods [16, 29], but come at the cost of significantly reduced efficiency due to unfavorable term weight distributions [19, 21]. Furthermore, while a number of LSR approaches have been proposed [29], more effective models tend to result in slower query processing, resulting in a rich Pareto frontier of possible methods [3, 15, 16, 39].

Query Processing. To efficiently process queries over inverted indexes, *dynamic pruning* algorithms are typically employed [4, 11, 23, 37]. These algorithms make cheap estimations of document scores *before* undertaking any scoring operations, and avoid scoring documents that cannot disrupt the top-k "seen so far" documents. This is achieved by keeping track of the highest impact posting for each postings list offline (to estimate the score of the given candidate document), and the lowest scoring document in the current top-k min-heap (as the threshold for entry into the top-k). We refer the reader to the survey of Tonellotto et al. [36] for a comprehensive overview of dynamic pruning algorithms.

Recent work has investigated the use of dynamic pruning algorithms in the context of LSR. The common consensus is that factors including the expansion of queries and terms generated by LSR models – as well as the non-standard term distributions – causes significant decreases in efficiency compared to traditional rankers [21]. In response, various efforts have focused on improving the efficiency of query processing in the LSR regime [6, 9, 15, 19, 22, 25, 32, 40]. However, these works all assume *static* and *offline* indexing.

Operational Context. Figure 1 presents the operational context we assume within this work. In particular, it is important to distinguish the role of the dynamic index from the much larger, static index. The role of the dynamic

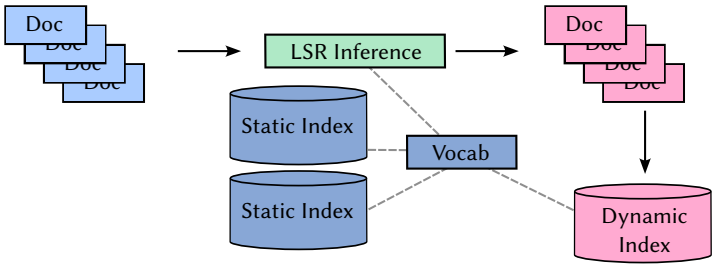

Fig. 1: The operational context assumed in this work. New documents arrive at any time and are processed by an LSR model; this model has access to global collection statistics and a shared vocabulary. After the documents have been processed by the LSR model, they can be ingested into the dynamic index.

index is to serve the *fresh* tier of incoming documents, allowing them to be immediately queried. Once the dynamic index becomes sufficiently large, it will be offloaded into the static index, and a new, empty dynamic index will resume the processing of the fresh tier. Given this configuration, we assume that the LSR model has access to the vocabulary of the static indexes, and that this vocabulary is shared across all index shards.

3 Learned Sparse Linked Block Index

Now, we describe the *linked block* index, and the adaptations we make such that the index can be applied to the learned sparse scoring regime.

Index Overview. The linked block index is an *immediate access* index, meaning that documents can be queried as soon as they are ingested [26]. The index is stored in a single contiguous slab of memory, the entirety of which is allocated at the beginning of indexing.[1] Each posting list takes the form of a linked list of fixed-sized *blocks*. A hash table representing the index vocabulary provides constant-time access to the first (head) block of each postings list. As blocks become full, new blocks are appended to the next free space of the memory slab and are linked by an implicit pointer, resulting in many interleaved lists of blocks. Figure 2 shows this structure. Each postings list is made up of three block types, shown in Figure 3:

– *Head block:* The head block stores important metadata such as the number of postings (f_t), the last document identifier written to this postings list, the write-offset in the tail block, and the most recent document added to the index. The remaining unused space of the block is used to store postings.

[1] We assume that a fixed memory budget governs the size of the index, hence pre-allocating the memory slab. Allocation can be done on-demand [27].

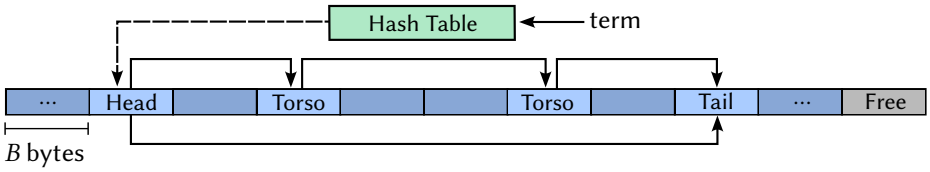

Fig. 2: An overview of the *linked block* index structure. A hash table maps terms to head blocks that maintain postings metadata. Each block is a fixed size, B bytes, and four of those bytes store an implicit pointer to the next block in the sequence, The head block also stores a pointer to the final (tail) block in the sequence for efficient access to write new postings. The index is of a fixed total size, and may contain free (unused) blocks at the end. Adapted from [26].

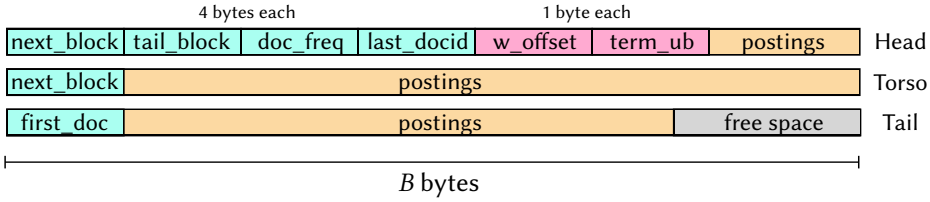

Fig. 3: An overview of the modified block types. Each block is a fixed size, B bytes. The head block stores metadata including a pointer to the next block, a pointer to the tail block, the document frequency of this term, the last document identifier indexed in this list, the write offset position within the tail block, and the term-wise impact upper bound score. Adapted from [26].

- *Tail block:* The tail block is the final block of the linked postings list. The tail block stores the identifier of the first document added to the block (used later for calculation of between-block gaps). New postings are added to the next free space of the tail block; and
- *Torso blocks:* When a tail block has no more space for postings, it is converted to a torso block by overwriting the first_doc field with an offset to the (current) tail block, the next block in the sequence. A torso block has a pointer to the next block, and is otherwise completely filled with postings.

It is worth noting that, since document identifiers are delta-coded, the *first posting* encoded in all torso or tail blocks stores a *block gap* representing the difference between the first identifier in the current block, and the first identifier in the previous block. This allows iteration across whole blocks *without* the need to decode all of the elements within each block, and is critically important for maintaining efficient query processing. It is also worth noting that since the write offset variable is a byte, the maximum block size is $B = 256$.

3.1 Modifications

The original implementation of the linked block index used a novel *Double VByte* compression codec that compressed both the document identifier and term frequency pair into a single element in the postings list. However, in LSR, term frequencies (which are typically very small) are replaced with term *impacts* that are usually in the range $[1, 255]$ (such that they can be stored within a single byte). These much larger payload values disrupt the effectiveness of the double VByte approach. Instead, we apply the standard VByte algorithm on the document identifiers [35], and to store impacts as a single uncompressed byte (since VByte cannot possibly encode them into a smaller value).

Our second key modification adjusts the head block of each posting list to store the *upper-bound impact score* for the list. This adjustment is vital to accommodate the use of dynamic pruning algorithms such as WAND or MaxScore [5, 37]. Moffat and Mackenzie [26] briefly mentioned this idea, stating that an additional *4-byte field* would be required. However, we guarantee that impact scores are bounded by 255, meaning only a single byte is required per postings list (Figure 3). As will become clear later, this (minuscule) overhead is integral to supporting fast query processing. The upper-bound is checked and updated as new postings are inserted.

Finally, our operating context assumes a shared vocabulary across the static and dynamic indexes. This means we can represent terms by their internal integer representations, rather than as raw strings. This change means that mapping a term identifier to a head block can be done via perfect hashing, and that the head block no longer needs to store the associated string. As a consequence, queries and documents are pre-processed to convert strings to their internal number representations. If a new, unseen term occurs, we assume that the shared vocabulary adds it as a new entry, and assigns the next logical term identifier, meaning that the dynamic index does not need to deal with strings at all.

4 Experimental Setup

This section describes the experimental settings used in our empirical evaluation. All experimental resources are made publicly available to facilitate reproducibility.[2]

4.1 Hardware and Systems

All experiments are executed on a single core of an AMD Ryzen Threadripper PRO 5975WX with 512GiB of main memory. The system runs Ubuntu Linux, and all software was compiled using `gcc 11.4.0` with `-O3` optimizations enabled.

The state-of-the-art PISA system was used to represent a fast, compact, but *inflexible* indexing baseline; its index size and query latency represent idealistic targets, but it cannot dynamically index new documents on-the-fly. PISA was

[2] Please see the repository here: https://github.com/JMMackenzie/lslb/

Table 1: Summary statistics of the full 8.8 million passage index and associated query set for the four rankers under consideration. Space reports the size of the `json` file representing the collection in GiB.

Ranker	Space	Vocabulary	Postings		Query Length	
			Total	Per Doc.	All	Uniq.
BM25	2.97	2,660,824	266,247,718	30.1	4.5	3.4
DocT5Query	4.77	3,929,111	452,197,951	51.1	4.5	3.4
DeeperImpact	7.76	3,385,352	742,223,716	83.9	5.9	4.9
SPLADEv3	16.61	27,952	1,483,571,102	167.8	915.3	24.1

configured to use the SIMD-BP128 compression codec [17], representing a good balance between index space occupancy and query latency [24]. Similarly, the Lucene search engine was used (via the Anserini toolkit [38]) to represent a modern "production ready" search system, and is used widely in industrial settings [13]. Since Lucene indexes directly from disk, all experiments using Lucene load the `json` data blobs from a 2TiB solid state drive with read speeds of up to 6GiB/second to avoid disk overhead influencing the measurement. It is worth noting that Lucene indexing and retrieval has been heavily optimized for learned sparse retrieval, employing vectorization, compression codecs tailored to impact-based indexes, and document re-ordering by default [20].[3] These optimizations are not applied to either PISA or LinkedBlock in this work.

4.2 Documents and Queries

Our experiments are conducted on the `MSMARCO-v1` passage collection [2], consisting of around 8.8 million passages. We use the associated set of 6,980 `dev` queries for our efficiency experimentation.

Table 1 provides an overview of the properties of the collection after it was indexed with each specified ranker. Since each ranker involves different pre-processing operations such as document expansion or inference to generate per-document impact scores, the statistics of the downstream representations vary even though the input corpus is the same. Similarly, queries are treated according to the ranking model being used; SPLADEv3 notably generates much longer queries, as query expansion and term weighting are essential to achieving strong effectiveness [15, 16].

To experiment with varying index sizes, we generate subsets of each index containing $x \times 10^y$ documents, using all combinations of $x \in \{1, 5\}$ and $y \in \{4, 5, 6\}$. All document collections are represented as `json` blobs, where each record (representing a document) contains a list of term/weight pairs, and these represent the starting point of our experiments.

[3] See, for example: https://lucene.apache.org/core/corenews.html

Retrieval Sizes. Top-k retrieval systems may be used for different tasks, including the direct retrieval of results to end users (where k is small), or perhaps as first-stage candidate generation systems (where k is large). To represent these settings of interest, we experiment with values of $k \in \{10, 1000\}$.

Algorithms. Learned sparse retrieval systems score documents according to the "sum of weights" across query terms, meaning that *disjunctive* matching semantics are commonly used. To this end, we experimented with two document-at-a-time query processing algorithms: the naïve DAAT traversal (where all postings across all query terms are processed); and the MaxScore algorithm [37].

Models. In order to compare indexing and querying performance over both traditional and LSR models, we employ two representatives from each family. Both traditional approaches use BM25 scoring [33]; one is applied to the original corpus (denoted BM25), and the other is applied to an index that has been augmented with queries from the DocT5Query model [30] (also denoted BM25-T5). Impact scores are pre-computed and stored as 8-bit quantities. Two representative LSR models were used: DeeperImpact [3], which does not require query-time inference to encode the input query [28], and SPLADEv3 [16] which *does* require query-time inference to expand and weight the query terms.[4] Both are representative of the current state-of-the-art (Pareto optimal) LSR models.

5 Experiments and Analysis

In this section, we outline our empirical experimentation and subsequent analysis. We focus on three distinct aspects, all with the aim of exploring how the current state-of-the-art LinkedBlock index generalizes to learned sparse retrieval. Firstly, we explore the index size of the LinkedBlock system; secondly, we examine the indexing throughput of the LinkedBlock system; finally, we examine the query processing performance of the LinkedBlock system. In each aspect, we compare LinkedBlock to existing techniques, where applicable, including both PISA and Lucene, as well as with the reported figures from the original LinkedBlock work [26], and the Apoptosic index [12].

Index Size. Our first experiment measures the disk size of the LinkedBlock system across the four models. Static indexes are capable of more efficient compression techniques, such as block-based compression [17, 31], that our dynamic index cannot use. As such, we expect that our index will not perform as well as the baseline static indexes in this aspect, though our performance should remain competitive. It should be noted when our index is written to disk, the vocabulary table is written alongside it, consuming a significant amount of space – up to 15% of the total index space in the worst case.

Table 2 shows the size in MiB of our index with two block sizes, two collection sizes, and across the four ranking models. Unsurprisingly, LinkedBlock performs worse than the static PISA and Lucene systems. However, LinkedBlock

[4] Query inference is not counted in the latency, as we use pre-encoded queries.

Table 2: Index size, in MiB, for LinkedBlock with two different block sizes across two of the index subset sizes (measured in documents). Both the PISA and Lucene numbers are provided as reference points. Although there is some headroom, LinkedBlock remains competitive with the baselines, especially on the LSR models.

Ranker	Subset: 1 million				Full collection: 8.8 million			
	PISA	Lucene	$B = 64$	$B = 128$	PISA	Lucene	$B = 64$	$B = 128$
BM25	91	83	117	150	725	703	828	949
DocT5Query	135	116	180	227	1092	997	1321	1496
DeeperImpact	200	173	257	296	1673	1499	1963	2068
SPLADEv3	373	337	393	374	3379	3032	3441	3265

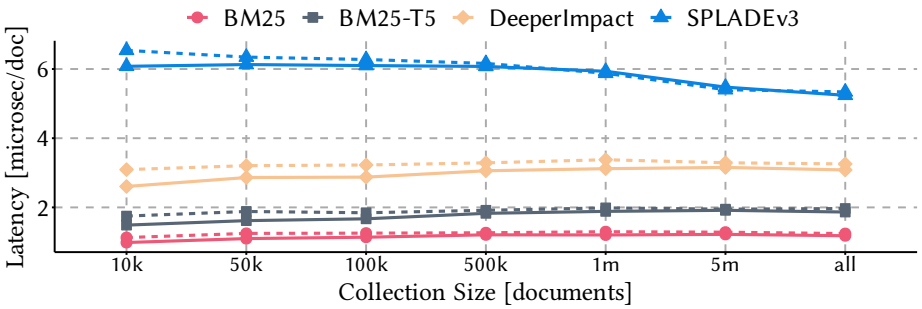

Fig. 4: Indexing throughput for LinkedBlock with $B = 64$ (solid line) and $B = 128$ (dashed line) as the index size grows for each of the four rankers. As expected, the LSR models take longer to index, per document, as the document representations contain more terms (see Table 1). Indexing throughput remains constant as the number of documents in the index increases.

is unexpectedly competitive with both static indexes when using the SPLADEv3 collection. A key characteristic of the SPLADEv3 collection is its relatively small vocabulary size. A potential waste of space in our index is terms that have very small number of postings, as a few bytes of such a posting list will be filled, but the rest will be left as empty space. A smaller vocabulary size means that we are much more likely to have heavily populated posting lists, leaving less chance for wasted space. Since we are also writing our vocabulary table to disk, a smaller vocabulary also saves a significant amount of space.

Indexing Throughput. Our next experiment measures the indexing throughput of the LinkedBlock system across the four models. Given the simplicity of the block structure, indexing should be extremely efficient, as the only operations required are to interrogate the hash table to find the head block, jump to the

Table 3: Average query response time, in milliseconds, for top $k = 10$ retrieval across two corpus sizes. PISA employs the MaxScore processing algorithm; Lucene uses a vectorized Block-Max MaxScore approach; and LinkedBlock ($B = 128$) uses both Naïve (LB-N) and MaxScore (LB-MS) processing.

Ranker	Subset: 1 million				Full collection: 8.8 million			
	PISA	Lucene	LB-N	LB-MS	PISA	Lucene	LB-N	LB-MS
BM25	0.25	0.72	2.09	0.71	1.18	2.45	18.6	4.24
DocT5Query	0.11	1.50	10.6	0.59	0.44	4.99	97.8	5.74
DeeperImpact	4.14	3.86	20.3	10.9	28.6	14.8	192.6	90.7
SPLADEv3	5.93	6.91	24.2	12.1	35.9	36.1	215.0	83.3

tail block and write offset, and encode the document identifier and the impact, along with some minimal book-keeping.

Figure 4 reports the latency per document insertion across the different index sizes and ranking models. At a maximum of around 6 microseconds per document, our modified LinkedBlock structure can index over 100,000 documents per second – at least for the short passages used in these experiments. In wall-clock time, even the largest indexes can be built over the entire `MSMARCO-v1` passage collection with the SPLADEv3 documents in under 1 minute.

Lucene, on the other hand, takes between 100 and 500 microseconds per document (with the larger timing arising with the longer SPLADEv3 documents), taking up to 40 minutes to index the entire collection. As further reference points, the original LinkedBlock index was reported to take on the order of 10 microseconds per document [26], and the Apoptosic index runs at around 2 microseconds per posting (so, perhaps 50 to 100 microseconds per document).

We must explicitly acknowledge that comparing the production-ready Lucene engine to our experimental codebase is not an apples-to-apples comparison. For example, our experimental set-up assumes that a fixed slab of memory is available ahead of time to use for the LinkedBlock system; our code is written in C, and Lucene in Java; and we rely on the LSR models to pre-map terms to the vocabulary as shown in Figure 1. Furthermore, the true indexing latency of the PISA baseline is difficult to estimate, as the current pipeline uses multiple, separate stages. Nonetheless, our results demonstrate that the LinkedBlock approach is highly competitive to production-ready solutions, even for somewhat "large" indexes in the context of handling append-only document corpora.

Query Latency. Next, we turn our attention to query processing latency. Table 3 shows the cost of query processing across the different systems employed. This experiment examines two collection sizes, and fixes the number of retrieved results, k, to 10 (with similar trends observed when $k = 1{,}000$). Interestingly, all systems perform well for the quantized BM25 indexes (BM25 and DocT5Query), requiring just a few milliseconds to retrieve the top ten documents on these

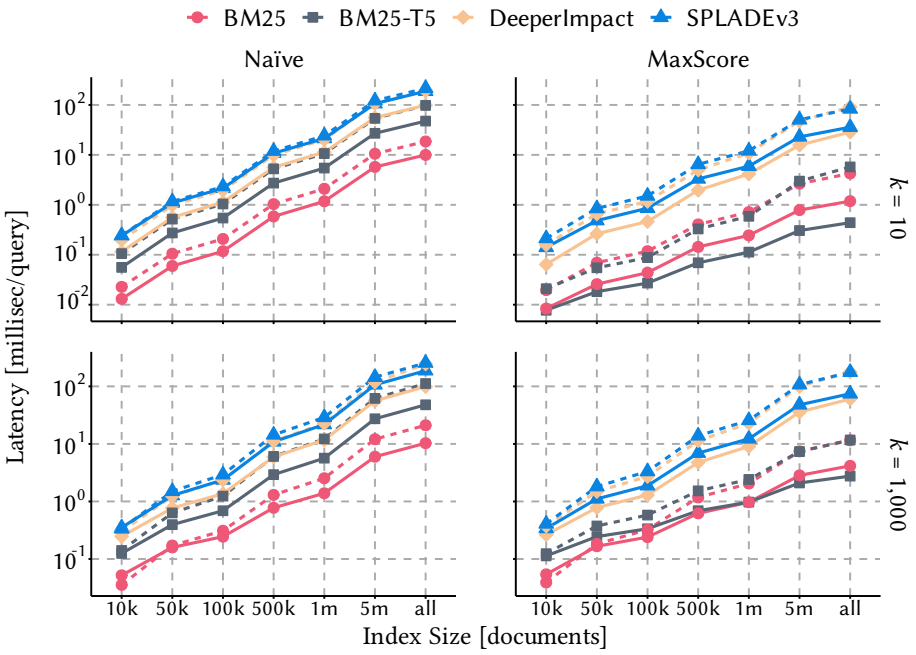

Fig. 5: Query processing latency (mean response time) for PISA (solid line) and LinkedBlock with $B = 128$ (dashed line) as the index size grows across the four rankers. The results for $B = 64$ are very similar to those with $B = 128$ and are omitted for clarity. Note the logarithmic vertical axis.

collections. The only exception is the Naïve algorithm, which processes every posting, and thus has runtime that grows roughly linearly with the number of postings under consideration. This effect is clear when comparing the original index (BM25) with the expanded index (DocT5Query).

The story changes when we consider the two LSR models, though, with query latency in the tens or hundreds of milliseconds on the full collection. Both PISA and Lucene clearly outperform LinkedBlock, which takes between 2 to 6× longer for the optimized MaxScore traversal. Given that both PISA and Lucene are highly optimized, and that the LinkedBlock index supports dynamic indexing, this trade-off is not surprising. Nonetheless, the latency of LinkedBlock is still well within acceptable, especially on the smaller 1 million document subset, where queries can be answered within around 12 milliseconds.

Finally, considering LinkedBlock in isolation, it is interesting to see how much of a positive influence the dynamic pruning MaxScore method has over the Naïve approach. We observe speedups of between 4–14× for the traditional rankers, and between 2–2.6× for the LSR models.

Figure 5 provides a more comprehensive view on the query processing results, using PISA as a representative to which the LinkedBlock index is contrasted.

In particular, both the naïve document-at-a-time algorithm and MaxScore are presented (vertical slices) across two values of k (horizontal slices).

From this figure, a few new findings are evident. Firstly, the solid and dashed lines are more or less parallel across the different settings (ranker, k, query processing scheme) as the size of the index increases. This suggests that the LinkedBlock index *scales as well as* the static PISA index. Secondly, we can observe that the LinkedBlock index is highly efficient when the collection size remains small. Since this type of system is deployed is to handle the *fresh document tier*, we expect that the LinkedBlock index should not be expected to handle huge quantities of documents; and the fact that it is still empirically performant with even one *million* documents makes it a good choice in practice.

Generalizability of the Enhancements. Our final experiment involved taking the original reference implementation of Moffat and Mackenzie [26], and incorporating the key enhancements we have described above, to determine whether they generalize to the original system. That is, we switched their *Double VByte* compression to use the simple *VByte* codec on the document identifiers, storing the impacts directly as bytes; we incorporated the term-wise upper-bound scores into the head block; and we implemented the MaxScore algorithm for fast top-k query processing. We did not change the vocabulary structure of the original implementation, so terms are explicitly stored in the head blocks.

Experiments on the indexing component were slightly slower (within one or two microseconds) of our C implementation. Similarly, the reference indexes were slightly larger than ours (within 5%), as they incur a minor overhead for explicitly storing each term. Query processing, on the other hand, was slightly faster in the reference implementation, with MaxScore processing over the full collections running in around 5 milliseconds for the traditional indexes (on par with our implementation), and approximately 75 milliseconds for the LSR indexes (compared to 83-90 milliseconds in Table 3). This both validates our findings, and affirms that the LinkedBlock index does indeed generalize to learned sparse retrieval.

6 Conclusion and Future Work

In this work, we revisited the problem of dynamic indexing in the context of learned sparse retrieval systems. We have demonstrated that state-of-the-art dynamic indexes such as LinkedBlock are capable of performing competitively with static indexes in the context of learned sparse retrieval. We have proposed extensions to the original proposed LinkedBlock structure to facilitate the use of the dynamic pruning algorithms like MaxScore, and that said algorithms provide significant improvements in query latency. Our experiments also showed that dynamic indexes scale as well as static indexes as collection/index size grows. Finally, integrating our extensions into the original codebase confirmed their effectiveness, and validated our analysis.

Although we have adapted the LinkedBlock structure for use with LSR, we have not made any changes to address the efficiency shortcomings of the

LSR paradigm when compared to traditional methods. In future work, it would be worth exploring further optimizations to LinkedBlock, including block-based dynamic pruning (like BMW [11]), or LSR-tailored compression schemes. Similarly, revisiting other efficient *static* LSR systems in the context of dynamic document streams could lead to even more practical systems.

Acknowledgments. We thank Alistair Moffat for some preliminary discussions and feedback. We also thank the anonymous referees for their feedback and suggestions. This project was supported by a Google Research Scholar grant.

Disclosure of Interests. The authors have no competing interests of any sort.

Bibliography

[1] Anh, V.N., de Kretser, O., Moffat, A.: Vector-space ranking with effective early termination. In: Proc. SIGIR, pp. 35–42 (2001)
[2] Bajaj, P., Campos, D., Craswell, N., Deng, L., Gao, J., Liu, X., Majumder, R., McNamara, A., Mitra, B., Nguyen, T., Rosenberg, M., Song, X., Stoica, A., Tiwary, S., Wang, T.: MS MARCO: A human generated machine reading comprehension dataset. arXiv:1611.09268 (2016)
[3] Basnet, S., Gou, J., Mallia, A., Suel, T.: Deeperimpact: Optimizing sparse learned index structures. In: Proc. SIGIR ReNeuIR Workshop (2024)
[4] Broder, A.Z., Carmel, D., Herscovici, M., Soffer, A., Zien, J.: Efficient query evaluation using a two-level retrieval process. In: Proc. CIKM, pp. 426–434 (2003)
[5] Broder, A.Z., Carmel, D., Herscovici, M., Soffer, A., Zien, J.: Efficient query evaluation using a two-level retrieval process. pp. 426–434 (2003)
[6] Bruch, S., Nardini, F.M., Rulli, C., Venturini, R.: Efficient inverted indexes for approximate retrieval over learned sparse representations. In: Proc. SIGIR, pp. 152–162 (2024)
[7] Busch, M., Gade, K., Larson, B., Lok, P., Luckenbill, S., Lin, J.: Earlybird: Real-time search at Twitter. In: Proc. ICDE, pp. 1360–1369 (2012)
[8] Büttcher, S., Clarke, C.L.A.: Indexing time vs. query time: Trade-offs in dynamic information retrieval systems. In: Proc. CIKM, pp. 317–318 (2005)
[9] Carlson, P., Xie, W., He, S., Yang, T.: Dynamic superblock pruning for fast learned sparse retrieval. In: Proc. SIGIR, pp. 3004–3009 (2025)
[10] Crane, M., Trotman, A., O'Keefe, R.: Maintaining discriminatory power in quantized indexes. In: Proc. CIKM, pp. 1221–1224 (2013)
[11] Ding, S., Suel, T.: Faster top-k document retrieval using block-max indexes. In: Proc. SIGIR, pp. 993–1002 (2011)
[12] Eades, P., Wirth, A., Zobel, J.: Immediate text search on streams using apoptosic indexes. In: European Conference on Information Retrieval, pp. 157–169, Springer (2022)
[13] Grand, A., Muir, R., Ferenczi, J., Lin, J.: From MaxScore to Block-Max Wand: The story of how Lucene significantly improved query evaluation performance. In: Proc. ECIR, pp. 20–27 (2020)

[14] Khattab, O., Hammoud, M., Elsayed, T.: Finding the best of both worlds: Faster and more robust top-k document retrieval. In: Proc. SIGIR, pp. 1031–1040 (2020)
[15] Lassance, C., Clinchant, S.: An efficiency study for SPLADE models. In: Proc. SIGIR, pp. 2220–2226 (2022)
[16] Lassance, C., Déjean, H., Formal, T., Clinchant, S.: SPLADE-v3: New baselines for SPLADE. arXiv preprint arXiv:2403.06789 (2024)
[17] Lemire, D., Boytsov, L.: Decoding billions of integers per second through vectorization. Soft. Prac. & Exp. **45**(1), 1–29 (2015)
[18] Lester, N., Moffat, A., Zobel, J.: Fast on-line index construction by geometric partitioning. In: Proc. CIKM, pp. 776–783 (2005)
[19] Mackenzie, J., Mallia, A., Moffat, A., Petri, M.: Accelerating learned sparse indexes via term impact decomposition. In: Proc. EMNLP (Findings) (2022)
[20] Mackenzie, J., Petri, M., Moffat, A.: Tradeoff options for bipartite graph partitioning. Trans. Knowledge & Data Eng. **35**(8), 8644–8657 (2023)
[21] Mackenzie, J., Trotman, A., Lin, J.: Efficient document-at-a-time and score-at-a-time query evaluation for learned sparse representations. ACM Trans. Inf. Syst. **41**(4) (2023)
[22] Mallia, A., Mackenzie, J., Suel, T., Tonellotto, N.: Faster learned sparse retrieval with guided traversal. In: Proc. SIGIR, pp. 1901–1905 (2022)
[23] Mallia, A., Ottaviano, G., Porciani, E., Tonellotto, N., Venturini, R.: Faster blockmax WAND with variable-sized blocks. In: Proc. SIGIR, pp. 625–634 (2017)
[24] Mallia, A., Siedlaczek, M., Suel, T.: An experimental study of index compression and DAAT query processing methods. In: Proc. ECIR, pp. 353–368 (2019)
[25] Mallia, A., Suel, T., Tonellotto, N.: Faster learned sparse retrieval with block-max pruning. In: Proc. SIGIR, pp. 2411–2415 (2024)
[26] Moffat, A., Mackenzie, J.: Efficient immediate-access dynamic indexing. Inf. Proc. & Man. **60**(3), 103248 (2023)
[27] Moffat, A., Mackenzie, J.: Immediate-access indexing using space-efficient extensible arrays. In: Proc. ACDS, pp. 2.1–2.8 (2023)
[28] Nardini, F.M., Nguyen, T., Rulli, C., Venturini, R., Yates, A.: Effective inference-free retrieval for learned sparse representations. In: Proc. SIGIR, pp. 2936–2940 (2025)
[29] Nguyen, T., MacAvaney, S., Yates, A.: A unified framework for learned sparse retrieval. In: Proc. ECIR, pp. 101–116 (2023)
[30] Nogueira, R., Lin, J.: From doc2query to docTTTTTquery (2019), URL https://cs.uwaterloo.ca/~jimmylin/publications/Nogueira_Lin_2019_docTTTTTquery-latest.pdf, unpublished report, David R. Cheriton School of Computer Science, University of Waterloo, Canada
[31] Pibiri, G.E., Venturini, R.: Techniques for inverted index compression. ACM Computing Surveys **53** (2020)
[32] Qiao, Y., Yang, Y., Lin, H., Yang, T.: Optimizing guided traversal for fast learned sparse retrieval. In: Proc. WWW, pp. 3375–3385 (2023)

[33] Robertson, S., Walker, S., Jones, S., Hancock-Beaulieu, M.M., Gatford, M.: Okapi at TREC-3. In: Proc. TREC, pp. 109–126 (1994)
[34] Robertson, S., Zaragoza, H.: The probabilistic relevance framework: BM25 and beyond. Found. Trends Inf. Ret. **3**, 333–389 (2009)
[35] Thiel, L., Heaps, H.: Program design for retrospective searches on large data bases. Information Storage and Retrieval **8**(1), 1–20 (1972)
[36] Tonellotto, N., Macdonald, C., Ounis, I.: Efficient query processing for scalable web search. Found. Trends Inf. Ret. **12**, 319–500 (2018)
[37] Turtle, H., Flood, J.: Query evaluation: Strategies and optimizations. Inf. Proc. & Man. **31**(6), 831–850 (1995)
[38] Yang, P., Fang, H., Lin, J.: Anserini: Reproducible ranking baselines using Lucene. J. Data and Inf. Quality **10**(4), 16:1–16:20 (2018)
[39] Yates, A., Lassance, C., MacAvaney, S., Nguyen, T., Lei, Y.: Neural lexical search with learned sparse retrieval. In: Proc. SIGIR-AP, pp. 303–306 (2024)
[40] Yu, P., Mallia, A., Petri, M.: Improved learned sparse retrieval with corpus-specific vocabularies. In: Proc. ECIR, pp. 181–194 (2024)
[41] Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Computing Surveys **38**(2), 6:1–6:56 (2006)