Forward Index Compression for Learned Sparse Retrieval

Sebastian Bruch[1][0000−0002−2469−8242] , Martino Fontana[2], Franco Maria Nardini[3][0000−0003−3183−334X], Cosimo Rulli[3][0000−0003−0194−361X], and Rossano Venturini[4][0000−0002−9830−3936]

[1] Northeastern University, Boston, USA, s.bruch@northeastern.edu
[2] University of Pisa, Italy, martino.fontana@studenti.unipi.it
[3] ISTI-CNR, Pisa, Italy, {name.surname}@isti.cnr.it
[4] University of Pisa, Italy, rossano.venturini@unipi.it

Abstract. Text retrieval using learned sparse representations of queries and documents has, over the years, evolved into a highly effective approach to search. It is thanks to recent advances in approximate nearest neighbor search—with the emergence of highly efficient algorithms such as the inverted index-based Seismic and the graph-based Hnsw—that retrieval with sparse representations became viable in practice. In this work, we scrutinize the efficiency of sparse retrieval algorithms and focus particularly on the size of a data structure that is common to all algorithmic flavors and that constitutes a substantial fraction of the overall index size: the forward index. In particular, we seek compression techniques to reduce the storage footprint of the forward index without compromising search quality or inner product computation latency. In our examination with various integer compression techniques, we report that StreamVByte achieves the best trade-off between memory footprint, retrieval accuracy, and latency. We then improve StreamVByte by introducing DotVByte, a new algorithm tailored to inner product computation. Experiments on MsMarco show that our improvements lead to significant space savings while maintaining retrieval efficiency.

1 Introduction

In recent years, Large Language Models (Llms) have been fruitfully employed to encode documents and queries into high-dimensional spaces while approximately preserving their pairwise semantic similarity. One attractive family of encoders or *embedding functions* produces *sparse* representations by mapping text into a space with tens of thousands of dimensions, but where most of the coordinates of the representation are zero. These techniques have gained much attention due two attributes: 1) *interpretability*, as the representation produced is grounded in the vocabulary of the collection; and, 2) *effectiveness*, as these techniques can heavily adopt query and document expansion so to actively counter the vocabulary mismatch problem at retrieval time.

Despite these properties, the use of sparse embeddings in text retrieval faced a practical challenge: efficiency of executing queries. Recently, however, several

efforts emerged in the literature that enable fast and effective (approximate) retrieval for such embeddings, rendering the paradigm viable in practice.

Among what the literature has to offer, the highly efficient algorithms are often adaptations of existing Approximate Nearest Neighbor Search (ANNS) algorithms [2] to sparse vectors (e.g., HNSW [16,18,9]), or ANNS algorithms designed specifically for learned sparse vectors (e.g., SEISMIC [3,4,5,6], BMP [17], DSBP [7]). Regardless of how they organize their index or perform ANNS, all such algorithms rely on a key data structure: the forward index.

The forward index is simply a mapping from document identifiers to sparse vectors. Denoting by $\mathcal{X}$ a forward index, its purpose can be succinctly stated as enabling the computation of the inner product $\langle q, x \rangle$ for a query q and document $x \in \mathcal{X}$. That is why this structure is so critical to all sparse ANNS algorithms: Algorithms use it to "error-correct" the retrieved set by re-ordering an initial set of candidates by exact inner product obtained from the forward index.

Storing the forward index is conceptually straightforward. Noting that a sparse vector $x \in \mathbb{R}^d$ has $\text{nnz}(x) \ll d$, where $\text{nnz}(x) = |\{x_i : x_i \neq 0\}|$ is the number of nonzero coordinates, it is more reasonable to store only the nonzero coordinates of x in the forward index. One way to do that is by storing three arrays: *components*, *values*, and *offsets*. The *components* array records the positions of nonzero coordinates of all vectors in $\mathcal{X}$ sequentially, while *values* similarly registers their nonzero values. The *offsets* array tracks, for each vector $x^{(i)}$, where its components and values are stored within the corresponding arrays. This design allows one to store components and values in contiguous chunks of memory, thereby optimizing data access patterns.

It should be apparent that the size of the forward index is dominated by components and values: The number of bits required per component is $\lceil \log_2 d \rceil$, which is typically rounded to the nearest primitive data type such as the commonly-used 16-bit or 32-bit integers. Values are typically stored as 16-bit floating point numbers, which turn out to be as effective as 32-bit ones.

Given its role in sparse ANNS algorithms and its substantial memory footprint, our work studies effective solutions for compressing the forward index. For concreteness, we conduct this study in the context of the state-of-the-art sparse ANNS algorithm, SEISMIC [3,4,5,6]. Our specific contributions are as follows:

- We empirically evaluate a range of integer compression techniques applied to the forward index in SEISMIC, and analyze how each algorithm trades off size for efficiency. Our experiments reveal that StreamVByte [14] offers a compelling balance between compactness and decoding speed;
- We introduce DotVByte, a new compression technique based on StreamVByte. DotVByte is optimized for high-performance inner product computation—a key operation at the core of SEISMIC and sparse retrieval methods in general. Experimental results on MSMARCO demonstrate that our proposed compression algorithm achieves a substantial reduction in memory usage while maintaining equivalent retrieval effectiveness; and, finally,
- We provide a highly optimized implementation of DotVByte and StreamVByte in Rust. Our code is made public within the SEISMIC library.

2 Compressing the Forward Index

We focus exclusively on compressing the components array where each component is a 16-bit integer. Per the literature on integer sequence compression [20], we represent a document by gaps between consecutive components. We begin by examining standard methods from the literature, then present our contribution.

Before presenting findings, we note that any permutation π may be applied to the components provided the same permutation is applied to query vectors. That includes a π that minimizes the gaps in a document's component sequence—a problem analogous to one encountered in inverted indexes where the objective is to reorder document identifiers to produce more compressible inverted lists.

A standard approach to finding π is the Recursive Graph Bisection (RGB) algorithm [10]. It models the problem as a bipartite graph with "query" and "data" vertices, and permutes the data vertices to minimize the logarithmic gaps between consecutive neighbors in adjacency lists. The solution is obtained through recursive graph bisection, an iterative process that partitions the vertex set into two equal-sized subsets that optimize an estimate of the compressed size. In our formulation, components are data vertices and documents are query vertices. We use an open-source implementation of RGB by [15].[5]

2.1 Standard methods

Table 1 shows the performance of several standard compression techniques (with and without RGB) applied to our problem on the SPLADE embeddings of MS-MARCO. It presents the average bits-per-component and the average time ("Scan Time") required to compute the inner product between every document in MS-MARCO and 100 randomly-sampled queries from the `dev-small` set. We use the random-access implementation provided by the `compressed-intvec` Rust package[6] for all the methods except for StreamVByte, which we implement ourselves.

Algorithms. Before we discuss the results, let us briefly summarize two of the algorithms most relevant to this work. **VByte** [23] encodes an integer x using $L+1$ bytes, b_0 to b_L, where the most significant bit of each b_i serves as a control bit indicating whether the integer terminates in b_i (0) or continues in b_{i+1} (1). Decompressing x follows the formula $\sum_{i=0}^{L}(b_i \bmod 2^7) \cdot 2^{7i}$, where the modulo operation discards the control bits and the multiplication rescales the contribution of each byte.

StreamVByte [14] extends VByte by taking a page from VARINT-GB [8] and VARINT-G8IU [22]. It accelerates decompression through two key optimizations. First, when compressing an integer value x, it uses a 2-bit control to record the number of bytes x is compressed into. In this way, the control information for four values can be stored in a single byte, separately from the data.

This organization of the control information gives way to the second optimization: Decompression of all four values using the `_mm_shuffle_epi8` SIMD

[5] Obtained from `https://github.com/jmmackenzie/enhanced-graph-bisection`.
[6] `https://github.com/lukefleed/compressed-intvec`.

Table 1. Effect of RGB on four compression techniques in terms of average number of bits-per-component and time to compute inner products between a query and every document in MsMarco, averaged over 100 randomly-sampled queries.

Method	Bits per Component		Scan Time (sec.)	
	No RGB	RGB	No RGB	RGB
Uncompressed	16.0	-	0.3	-
VByte [23]	11.8	9.6	9.9	7.1
Elias Gamma [11]	12.1	8.6	10.1	9.6
Elias Delta [11]	10.9	8.1	7.9	7.6
Zeta [1]	9.8	7.5	8.4	7.1
StreamVByte [14]	12.5	11.2	1.3	1.2

instruction. To that end, it reads 16 raw bytes into a SIMD register, and the "permutation" stored in the control byte shuffles them so that the register can be interpreted as an array of four 32-bit integers. Not all 16 bytes from the initial read may remain in the register: the raw data must be "scrolled" by the number of bytes that have not been discarded, which must be calculated separately.

Results. Applying RGB effectively reduces bits-per-component, with increased compression of up to 27% with Elias Gamma. It also yields improved decompression performance because smaller values are faster to decompress—a phenomenon that is particularly pronounced for VByte. StreamVByte has the fastest decompression speed, but its compression ratio is relatively suboptimal.

VByte's memory efficiency is rather surprising. Its application to inverted indexes, while enjoying fast decompression, achieves a compression ratio that is multiple times worse than other algorithms such as Zeta (cf. Table 11 in [20]).

2.2 DotVByte: our contribution

We now describe how we specialize StreamVByte for the components array compression problem. First, observe that components are 16-bit integers, implying that a single control bit is adequate. As a result, `_mm_shuffle_epi8` can decompress 8 values simultaneously into a 128-bit register. In addition, the number of raw bytes to scroll is already encoded in the control byte itself through `popcnt`.

The second difference lies in the implementation. StreamVByte is general-purpose and stores the decoded values into a buffer before reading them individually. In contrast, we design DotVByte for maximum query performance by executing a sequence of operations directly in SIMD registers to compute inner products: a) decompress the components; b) gather the query values in the corresponding positions; c) read document values, convert them to `f32`, multiply them with the gathered query values, and add the packed product to the result.

Finally, in StreamVByte, a control byte can be shared between two consecutive documents. As a result, we are forced to decompress some values from the previous document, introducing unwanted overhead. We, however, enforce per-

document alignment: because the number of components encoded is a multiple of 8, all remaining components are stored uncompressed and processed normally.

3 Experimental Evaluation

We now evaluate DotVByte and baselines on MsMarco encoded using two state-of-the-art sparse embedding models: 1) Splade [12], widely used in retrieval efficiency benchmarks, and 2) Learned Inference-Free (LiLsr) [19].

LiLsr belongs to a family [13,21] that forgo query embedding and assign precomputed scores to query terms. The absence of query expansion, however, necessitates greater document expansion [19,13,21]: Splade embeddings of MsMarco have 119 (43) nonzero terms per document (query) while LiLsr embeddings have 387 (6). This 3.2× increase in the number of nonzero terms presents a challenge to compression and inner product computation efficiency.

Application to Seismic. Seismic is an efficient Anns algorithm designed for sparse embeddings. It works by organizing inverted lists into blocks, each with its own "summary" vector. During query processing, blocks are evaluated using the summary vector and, if the block is flagged, the exact inner product between the query and every document in that block is computed using the forward index.

We test compression algorithms using the publicly available implementation of Seismic.[7] For each dataset, we select the best index with the constraint that the overhead caused by the inverted index and summaries does not exceed 50% the size of the collection—reported as memory budget of 1.5× in [4,6]. On the resulting index, we tune query hyperparameters: $\mathsf{heap\_factor} \in \{0.7, 0.8, 0.9, 1.0\}$ and $\mathsf{cut} \in \{2, 4, 6, 7, 8, 10, 12\}$.

Hardware. Our code is compiled using Rust 1.94.0 (nightly). Experiments were conducted on a server equipped with an Intel(R) Core Ultra 7 CPU with 124 Gbyte of RAM with 20 cores. Search is performed using a single thread.

Results. Our experiments indicate that the scan time of DotVByte is 0.36 seconds (merely 20% slower than Uncompressed) and that it yields 10.8 bits per component under the same conditions of Table 1 (with RGB). In comparison to StreamVByte, it is more than 3× faster with an even smaller memory footprint.

Table 2 summarizes our results by average query latency (μsec.) at four different accuracy levels (i.e., the percentage of true nearest neighbors recalled), three different compression techniques for components, and using `float16` and `fixedU8`[8] representation for values. While our contribution focuses on component compression, we show that it is possible to halve the storage space required for the values with minimal performance degradation. Additionally, we report the space needed (in GB) to store the inverted index and forward index.

Among compressed indexes, Zeta yields the highest query latency but the largest reduction in space usage. StreamVByte trades off space for speed, with an average query latency which is about three times greater than Uncompressed.

[7] `https://github.com/TusKANNy/seismic`

[8] With the `fixed` Rust package: `https://docs.rs/fixed/latest/fixed/`.

Table 2. Mean latency (μsec) and index size (GB). Values are `float16` or `fixedU8`—with the latter, no solution reaches 99% accuracy given the memory budget.

Splade	`float16`					`fixedU8`				
Accuracy (%)	90	95	97	99	GB	90	95	97	99	GB
Uncompressed	157	289	387	1,081	5.8	185	290	485	-	4.8
Zeta	848	1,616	2,343	6,243	4.8	825	1,702	2,881	-	3.8
StreamVByte	425	818	1168	3146	5.2	402	768	1288	-	4.2
DotVByte	165	309	412	1082	5.1	156	305	482	-	4.2
LiLsr	`float16`					`fixedU8`				
Accuracy (%)	90	95	97	99	GB	90	95	97	99	GB
Uncompressed	518	773	773	1,658	17.5	670	734	734	-	14.3
Zeta	8,220	12,093	12,093	25,401	13.2	11,363	11,889	13,196	-	10.1
StreamVByte	2,019	3,123	3,123	6,493	15.4	2,386	2,559	2,943	-	12.2
DotVByte	529	788	788	1,643	14.9	662	728	822	-	11.7

DotVByte outperforms others in the latency-size trade-off space. In fact, it achieves a space reduction of up to 22% over Uncompressed (5% to 10% overhead with respect to Zeta) with similar query latency, both for Splade and LiLsr.

4 Conclusions and Future Work

We examined the efficiency of sparse Anns algorithms with a particular focus on the forward index—a substantial contributor to index size. By exploring a suite of integer compression techniques, we identified StreamVByte as providing the most favorable balance between memory usage, retrieval accuracy, and latency.

We then introduced an enhanced variant of StreamVByte, dubbed DotVByte, that is specifically optimized for inner product. Experiments on MsMarco confirm that it achieves significant space savings while preserving search quality and efficiency. These results highlight the potential of targeted compression strategies to improve the practicality of learned sparse retrieval systems.

For future work, we plan to incorporate sub-byte capability into DotVByte for small, frequent dgaps, to further improve the compression ratio.

Acknowledgements This work was partially supported by the Horizon Europe RIA "Extreme Food Risk Analytics" (EFRA), grant agreement No. 101093026, by the PNRR - M4C2 - Investimento 1.3, Partenariato Esteso PE00000013 - "FAIR - Future Artificial Intelligence Research" - Spoke 1 "Human-centered AI" funded by the European Commission under the NextGeneration EU program, and by the MUR-PRIN 2022 "Algorithmic Problems and Machine Learning", grant agreement n. 20229BCXNW.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Boldi, P., Vigna, S.: Codes for the World Wide Web. Internet mathematics **2**(4), 407–429 (2005)
2. Bruch, S.: Foundations of Vector Retrieval. Springer Nature Switzerland (2024)
3. Bruch, S., Nardini, F.M., Rulli, C., Venturini, R.: 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. pp. 152–162 (2024)
4. Bruch, S., Nardini, F.M., Rulli, C., Venturini, R.: Pairing clustered inverted indexes with knn graphs for fast approximate retrieval over learned sparse representations. In: Proceedings of the 19th ACM Conference on Information and Knowledge Management. pp. 3642–3646 (2024). https://doi.org/10.1145/3627673.3679977
5. Bruch, S., Nardini, F.M., Rulli, C., Venturini, R.: Efficient sketching and nearest neighbor search algorithms for sparse vector sets (2025), https://arxiv.org/abs/2509.24815
6. Bruch, S., Nardini, F.M., Rulli, C., Venturini, R., Venuta, L.: Investigating the scalability of approximate sparse retrieval algorithms to massive datasets. In: Proceedings of the 47th European Conference on Information Retrieval, ECIR. pp. 437–445 (2025)
7. Carlson, P., Xie, W., He, S., Yang, T.: Dynamic superblock pruning for fast learned sparse retrieval. In: Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval. pp. 3004–3009 (2025)
8. Dean, J.: Challenges in building large-scale information retrieval systems: invited talk. In: Proceedings of the Second International Conference on Web Search and Web Data Mining. p. 1 (2009)
9. Delfino, L., Erriquez, D., Martinico, S., Nardini, F.M., Rulli, C., Venturini, R.: kannolo: Sweet and smooth approximate k-nearest neighbors search. In: Proceedings of the 47th European Conference on Information Retrieval. pp. 400–406 (2025)
10. Dhulipala, L., Kabiljo, I., Karrer, B., Ottaviano, G., Pupyrev, S., Shalita, A.: Compressing graphs and indexes with recursive graph bisection. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 1535–1544 (2016)
11. Elias, P.: Universal codeword sets and representations of the integers. IEEE Transactions on Information Theory **21**(2), 194–203 (1975)
12. Formal, T., Lassance, C., Piwowarski, B., Clinchant, S.: 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. p. 2353–2359 (2022). https://doi.org/10.1145/3477495.3531857
13. Geng, Z., Ru, D., Yang, Y.: Towards competitive search relevance for inference-free learned sparse retrievers. CoRR **abs/2411.04403** (2024). https://doi.org/10.48550/ARXIV.2411.04403
14. Lemire, D., Kurz, N., Rupp, C.: Stream VByte: Faster byte-oriented integer compression. Information Processing Letters **130**, 1–6 (2018)
15. Mackenzie, J., Petri, M., Moffat, A.: Faster index reordering with bipartite graph partitioning. In: Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval. pp. 1910–1914 (2021)
16. Malkov, Y.A., Yashunin, D.A.: Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence **42**(4), 824–836 (4 2020)

17. Mallia, A., Suel, T., Tonellotto, N.: Faster learned sparse retrieval with block-max pruning. In: Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval. pp. 2411–2415 (2024)
18. Morozov, S., Babenko, A.: Non-metric similarity graphs for maximum inner product search. In: Advances in Neural Information Processing Systems (2018)
19. Nardini, F.M., Nguyen, T., Rulli, C., Venturini, R., Yates, A.: Effective inference-free retrieval for learned sparse representations. In: Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval. pp. 2936–2940 (2025)
20. Pibiri, G.E., Venturini, R.: Techniques for inverted index compression. ACM Computing Surveys **53**(6), 125:1–125:36 (2021)
21. Shen, X., Geng, Z., Yang, Y.: Exploring $\mathcal{L}_0$ sparsification for inference-free sparse retrievers. In: Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval. p. 2572–2576 (2025). https://doi.org/10.1145/3726302.3730192
22. Stepanov, A.A., Gangolli, A.R., Rose, D.E., Ernst, R.J., Oberoi, P.S.: Simd-based decoding of posting lists. In: Proceedings of the 20th ACM Conference on Information and Knowledge Management CIKM. pp. 317–326 (2011). https://doi.org/10.1145/2063576.2063627
23. Thiel, L.H., Heaps, H.: Program design for retrospective searches on large data bases. Information Storage and Retrieval **8**(1), 1–20 (1972)