Efficient and Effective Multi-Vector Dense Retrieval with EMVB

Franco Maria Nardini[1], Cosimo Rulli[1,*] and Rossano Venturini[2]

[1]ISTI-CNR, Pisa, Italy
[2]University of Pisa, Italy

Abstract

Dense retrieval techniques utilize large pre-trained language models to construct a high-dimensional representation of queries and passages. These representations assess the relevance of a passage concerning a query through efficient similarity measures. Multi-vector representations, while enhancing effectiveness, cause a one-order-of-magnitude increase in memory footprint and query latency by encoding queries and documents on a per-token level. The current state-of-the-art approach, namely PLAID, has introduced a centroid-based term representation to mitigate the memory impact of multi-vector systems. By employing a centroid interaction mechanism, PLAID filters out non-relevant documents, reducing the cost of subsequent ranking stages. This paper [1] introduces "Efficient Multi-Vector dense retrieval with Bit vectors" (EMVB), a novel framework for efficient query processing in multi-vector dense retrieval. Firstly, EMVB utilizes an optimized bit vector pre-filtering step for passages, enhancing efficiency. Secondly, the computation of centroid interaction occurs column-wise, leveraging SIMD instructions to reduce latency. Thirdly, EMVB incorporates Product Quantization (PQ) to decrease the memory footprint of storing vector representations while facilitating fast late interaction. Lastly, a per-document term filtering method is introduced, further improving the efficiency of the final step. Experiments conducted on MS MARCO and LoTTE demonstrate that EMVB achieves up to a $2.8\times$ speed improvement while reducing the memory footprint by $1.8\times$, without compromising retrieval accuracy compared to PLAID.

Keywords

Dense Retrieval, Multi-Vector, Efficiency, Bit Vectors.

1. Introduction

The widely acknowledged capability of Large Language Models (LLMs) to model semantic and context has been extensively used in Information Retrieval. In Dense Retrieval, LLMs are used to encode documents and queries into d-dimensional vectors. This enables the modeling of document-query relevance using simple metrics like Euclidean distance. In this line, a successful strategy involves using multi-vector representations for documents and queries, where a d-dimensional vector is produced for each token in the text. In this context, the similarity between the query and the passage is measured using the so-called late interaction mechanism. This mechanism works by computing the sum of the maximum similarities between each term of the query and each term of a candidate passage. Although multi-vector representations enhance effectiveness, they come at the cost of increased computational burden, including a larger memory footprint and longer retrieval time.

[1]This paper is an extended abstract of Nardini *et al.* [1].

SEBD 2024: 32nd Symposium on Advanced Database Systems, June 23-26, 2024, Villasimius, Sardinia, Italy

*Corresponding author.

Various approaches have been proposed to enhance the efficiency and reduce memory demands in multi-vector systems. ColBERT [2] exploits an inverted index to store all the terms embeddings and retrieve the candidate passages, but it necessitates maintaining the full-precision representation of each document term in memory, which can be substantial (e.g., 140 GB for MSMARCO). ColBERTv2 [3] introduces a centroid-based compression technique where each embedding is stored by saving the id of the closest centroid and then compressing the residual (i.e., the element-wise difference) by using 1 or 2 bits per component. ColBERTv2 saves up to $10\times$ space compared to ColBERT but sacrifices retrieval efficiency requiring up to 3 seconds to perform query processing on CPU. PLAID [4] builds on the embedding compressor of ColBERTv2 and leverages the centroid-based representation to discard non-relevant passages (centroid interaction), thus performing the late interaction exclusively on a carefully selected batch of passages. PLAID allows for massive speedup compared to ColBERTv2, but its average query latency can be up to 400 msec. on CPU with single-thread execution [4].

This paper introduces EMVB, a novel framework designed for efficient query processing in multi-vector dense retrieval. The key focus is on addressing the most time-consuming steps identified in PLAID, which include: i) extracting the top-nprobe closest centroids during candidate passage selection, ii) computing the centroid interaction mechanism, and iii) decompressing quantized residuals. To address the first two steps, we propose a highly efficient passage filtering approach based on optimized bit vectors. This approach significantly reduces the cost of top-*nprobe* extraction by identifying a small set of crucial centroid scores. Additionally, it decreases the number of passages for which centroid interaction computation is necessary. We further enhance efficiency in the second step by introducing a highly efficient column-wise reduction leveraging SIMD instructions. For the third step, late interaction efficiency is improved by introducing Product Quantization (PQ) [5]. This method provides comparable or superior performance compared to PLAID's bitwise compressor, while being up to $3\times$ faster. Additionally, we introduce a dynamic passage-term-selection criterion for late interaction, reducing the cost of this step by up to 30%.

Experimental evaluations on MS MARCO [6] passage (in-domain) and LoTTE [3] (out-of-domain) datasets demonstrate the effectiveness of EMVB compared to PLAID. On MS MARCO, EMVB achieves up to a $2.8\times$ speed improvement while reducing the memory footprint by $1.8\times$ without compromising retrieval accuracy. In the out-of-domain evaluation, EMVB delivers up to a $2.9\times$ speedup compared to PLAID with minimal loss in retrieval quality.

2. PLAID

In a multi-vector dense retrieval scenario, an LLM encodes a passage $\mathcal{P}$ into a collection of n_t dense d-dimensional vector T_j where n_t is the number of tokens in the passage. Encoding each token in each passage generates large collection, e.g., almost 600M of vectors for the 8.8M of passages in MSMARCO. In virtue of this, ColBERTv2 [3] and successively PLAID [4] exploit a centroid-based vector compression technique. First, the K-Means algorithm is used to identify a set of k centroids $\mathcal{C} = \{C_i\}_{i=1}^{n_c}$. The residual r between a vector x and its closest centroid $\bar{C}$ is computed so that $r = x - \bar{C}$ is computed, and then. compressed into $\tilde{r}$ using a b-bit encoder that represents each dimension of r using b bits, with $b \in \{1, 2\}$. This way, the memory footprint of each vector is given by $\lceil \log_2 |C| \rceil$ bits for the centroid index and $d \times b$ bits for the

compressed residual. At scoring time, decompressing the residual encoding is inefficient. For this reason, PLAID aims at decompressing as few candidate documents as possible by hinging on the so-called *centroid interaction* filtering step [4]. We now detail the PLAID retrieval system [4]. After the K-Means algorithm, each centroid is linked with a posting list containing the ids of the candidate passages. A passage belongs to a centroid C_i candidate list if at lest one of its tokens have C_i as its closest centroid. The query processing starts by computing the top-*nprobe* closest centroids for each query term q_i, with $i = 1, \ldots, n_q$, according to the *dot product* similarity measure. From the set of set of closest centroids, the candidates passages are retrieved, thanks to the previously built posting lists. In the centroid interaction step, the distance between the i-th query term q_i and a token embedding T_j with $j = 1, \ldots, n_p$ is computed as

$$q_i \cdot T_j \simeq q_i \cdot \bar{C}^{T_j} = \tilde{T}_{i,j}. \tag{1}$$

where $\bar{C}^{T_j}$ is the closest centroid to T_j. We estimate the score of a passage P with n_p terms as

$$\bar{S}_{q,P} = \sum_{i=1}^{n_q} \max_{j=1\ldots n_t} q_i \cdot \bar{C}^{T_j} \tag{2}$$

In the *decompression* phase the full-precision representation of P is reconstructed by combining the centroids and the residuals. Only the top-*ndocs* passages from the previous centroid interaction step move to this step. Finally, PLAID applied *late interaction* [2, 3] to computed the score of a re-constructed candidate passage against a query q. The late interaction measure is defined by Equation 3. Passages are then ranked according to their similarity score and the top-k passages are selected.

$$S_{q,P} = \sum_{i=1}^{n_q} \max_{j=1\ldots n_t} q_i \cdot T_j. \tag{3}$$

PLAID execution time. We present a detailed analysis of PLAID's execution time, delineating it into distinct phases such as *retrieval*, *filtering*, *decompression*, and *late interaction*. The experimentation adheres to the settings outlined in Section 4. The resulting execution times are reported for various values of k, representing the number of retrieved passages.

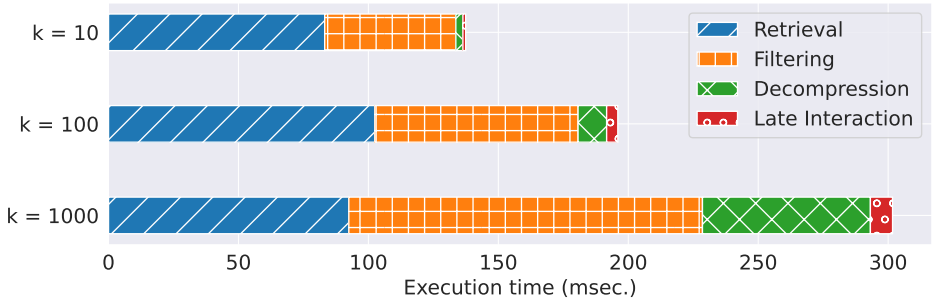

Figure 1: Breakdown of the PLAID average query latency (in milliseconds) on CPU across its four phases.

3. EMVB

Fast Closest Centroids Selection. The retrieval phase in PLAID is time consuming, as shown in Figure 1. This step concists of i) matrix multiplication between the query matrix and centroids for distance computation, ii) identify top-*nprobe* closest centroids, for each query term. Maybe surprisingly, the former step is the most time consuming ($3\times$ slower than matrix multiplication), even when performed with asymptotically linear algorithms such as *quickselect*. Our pre-filtering strategy, as explained in the subsequent paragraph, effectively accelerate the selection of the top-*nprobe* by minimizing the number of evaluated elements. In practice, we efficiently discard centroids with scores below a predefined threshold, and then exclusively apply quickselect to the remaining ones. As a result, in EMVB, the cost associated with extracting the top-nprobe becomes negligible, showcasing a speed improvement of two orders of magnitude when compared to extracting the top-nprobe from the complete set of centroids.

Pre-filtering using bitvectors. Let us recall the definition of $\tilde{T}_{i,j}$, which represents the approximate score of the j-th token of passage P with respect to the i-th term of the query q_i, as defined in Equation 1. Estimating whether $\tilde{T}_{i,j}$ has a large value is a proxy for estimating the importance of a passage P w.r.t to the query. Given a passage P, our pre-filtering consists in determining whether $\tilde{T}_{i,j}$, for $i = 1, \ldots, n_q$, $j = 1, \ldots, n_t$ is large or not. Recall that $\tilde{T}_{i,j}$ represents the approximate score of the j-th token of passage P with respect to the i-th term of the query q_i, as defined in Equation 1. Our pre-filtering approach works by checking if the centroid associated with T_j ($\bar{C}^T_j$) belongs to the set of the *closest centroids* of q_i. We define $\texttt{close}^{th}_i$ as the set of centroids whose scores surpass a specified threshold th in relation to a query term q_i. For a certain passage P, we also introduce the list of centroids ids I_P, where I^j_P is the centroid id of $\bar{C}^{T_j}$. The similarity of a passage with respect to a query can be rapidly estimated with our novel filtering function $F(P, q) \in [0, n_q]$ with the following equation:

$$F(P, q) = \sum_{i=1}^{n_q} \mathbf{1}(\exists\ j \text{ s.t. } I^j_P \in \texttt{close}^{th}_i). \tag{4}$$

For a passage P, this counts how many query terms have at least one similar passage term in P, where "similar" describes the belonging of T_j to $\texttt{close}^{th}_i$.

In Figure 2 (left), we present a performance comparison of our innovative pre-filter operating in conjunction with the centroid interaction mechanism (depicted by orange, blue, and green lines) against the performance of the centroid interaction mechanism applied to the entire set of candidate documents (indicated by the red dashed line) on the MS MARCO dataset. The plot illustrates that our pre-filtering approach efficiently eliminates non-relevant passages without adversely affecting the recall of the subsequent centroid interaction phase. For instance, we can significantly reduce the candidate passage set to just 1000 elements using a threshold of 0.4 without any compromise in R@100. In the subsequent sections, we detail the implementation of this pre-filter for optimal efficiency.

Building the bit vectors. Let $CS = q \cdot C^T$, with $CS \in [-1, 1]^{n_q \times |C|}$, with n_q is the number of query tokens, and $|C|$ is the number of centroids. For each i-th row of CS, we want to scan it and pick those j s.t. $CS_{i,j} > th$. This conceptually trivial algorithm can be implemented by leveraging SIMD instructions featured by modern CPUs. In particular, the AVX512 instruction set allows

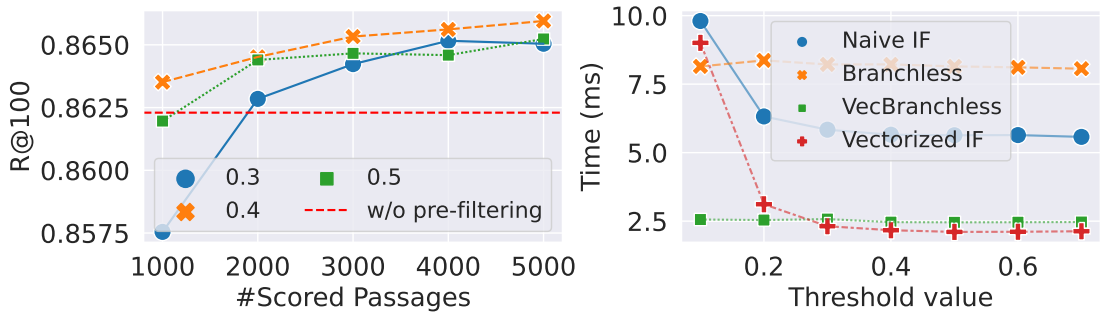

Figure 2: R@100 with various values of the threshold (**left**). Comparison of different algorithms to construct $\texttt{close}_i^{th}$, for different values of th (**right**).

to compare 16 fp32 values at a time thanks to the `_mm512_cmp_epi32_mask` instruction and store the comparison result in a $mask$ variable. Those indexes $J = \{j \in [0, 15] \mid mask_j = 1\}$ (if any) can be efficiently extracted by means of the `_mm512_mask_compressstore` instruction.

The effectiveness of algorithms employing if-based structures is largely contingent on the branch misprediction ratio. Contemporary CPUs speculate about the *if* condition's outcome by identifying patterns in the algorithm's execution flow. If an incorrect branch prediction occurs, a control hazard arises, leading to a pipeline flush with a delay of 15 to 20 clock cycles, approximately $10ns$. To address the inefficiency associated with branch misprediction, we introduce a branchless algorithm. For a detailed description of the algorithm and of its vectorized version, refer to the original paper [1].

Figure 2 (right) presents a comparison of our different approaches, namely "Naive IF," the "Vectorized IF," the "Branchless," and the "VecBranchless" described above. Branchless algorithms present a constant execution time, regardless of the value of the threshold, while if-based approaches offer better performances as the value of th increases. With $th \geq 0.3$, "Vectorized IF" is the most efficient approach, with a speedup up to 3 times compared to its naive counterpart.

Fast set membership. We now move to the problem of computing Equation 4, assuming $\texttt{close}_i^{th}$ to be known. Observe that this is a integer set membership problem, where we have to test if at least one member of I_P belongs to $\texttt{close}_i^{th}$, with $i = 1, \ldots, n_q$. *Bit vectors* (or bit array) are a widely adopted solution for implementing sets of integer values. A bit vector maps a set of integers up to N into an array of N bits, where the e-th bit is set to one if and only if the integer e is part of the set. Operations like addition and searching for any integer e can be executed in constant time using bit manipulation operators. In terms of memory occupancy, a bit vectors requires $N/8$ bytes. In our scenario, given that $|C| = 2^{18}$, a bit vector only needs $32K$ bytes for storage.

We further improve the efficiency of bit vectors by relying on the specific properties of our setting. As we need to search through all the n_q bit vectors at a time, we rearrange the representation of the bit vectors by stacking them vertically (Figure 3). This allows to search a centroid index through all the $\texttt{close}_i^{th}$ at a time. The bits corresponding to the same centroid for different query terms are consecutive and fit a 32-bit word. This way, we can simultaneously test the membership for all the queries in constant time with a single bitwise operation. In

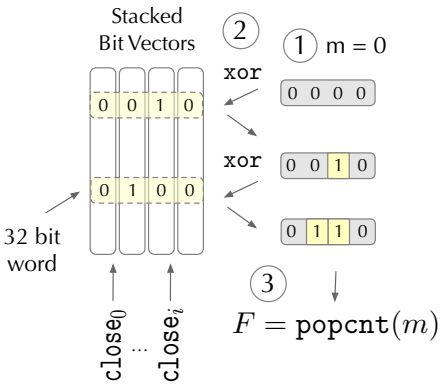

Figure 3: Vectorized Fast Set Membership algorithm based on bit vectors.

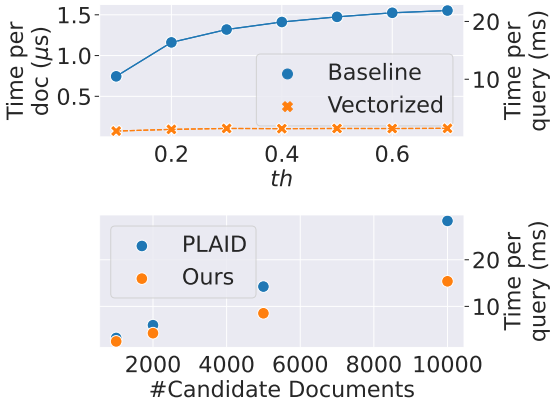

Figure 4: Vectorized vs naïve Fast Set Membership (**up**). Ours vs PLAID filtering (**down**).

detail, our algorithm starts by initializing a mask m of $n_q = 32$ bits at zeros (Step 1, Figure 3). Subsequently, for each term in the candidate documents, it performs a bitwise `xor` between the mask and the 32-bit word representing membership to all the query terms (Step 2, Figure 3). Consequently, Equation 4 can be derived by counting the number of 1s in m at the end of the execution using the `popcnt` operation available in modern CPUs (Step 3, Figure 3).

Figure 4 (up) showcases that our algorithm described in Figure 3 ("Vectorized") is $10\times$ to $16\times$ faster than the "Baseline" usage of bit vectors, and up to $30\times$ faster than the centroid-interaction proposed in PLAID [4], cf. Figure 4 (down).

Fast Filtering of Candidate Passages

Our pre-filtering approach allows us to efficiently filter out non-relevant passages and is employed upstream of PLAID's centroid interaction (Equation 2). The efficiency of the centroid interaction itself can be improved by using our column-wise reduction approach. For reason of space, we do not report the description of the algorithm in this discussion paper, and we encourage the reader to refer to the original work [1]. We implement PLAID's centroid interaction in C++ and we compare its filtering time against our SIMD-based solution. The results of the comparison are reported for different values of candidate documents in Figure 4 (down). Thanks to the proficient read-write pattern and the highly efficient column-wise max-reduction, our method can be up to $1.8\times$ faster than the filtering proposed in PLAID.

Late Interaction We propose to the b-bit residual compressor [3, 4] Product Quantization (PQ) [5]. Introducing PQ has two main advantages. On the one hand, it allows to compute the dot product between an input query vector q and the compressed residual r_{pq} without decompression. On the other hand, it allows to re-use the Consider a query q and a candidate passage P. We decompose the computation of the dot product between the query terms q_i and the closest centroid $\bar{C}^{T_j}$. Equation 3 becomes

$$S_{q,P} = \sum_{i=1}^{n_q} \max_{j=1\ldots n_t} (q_i \cdot \bar{C}^{T_j} + q_i \cdot r^{T_j}) \simeq \sum_{i=1}^{n_q} \max_{j=1\ldots n_t} (q_i \cdot \bar{C}^{T_j} + q_i \cdot r_{pq}^{T_j}), \tag{5}$$

where and $r^{T_j} = T_j - \bar{C}^{T_j}$. Our experimental evaluation shows that PQ is both faster (up to

$3.6\times$) and more effective compared to the b-bit compressor used in previous work. We propose to further improve the efficiency of the scoring phase by hinging on the properties of Equation 5. In many cases, we have that $q_i \cdot \bar{C}_j^T > q_i \cdot r_{pq}^{T_j}$; hence the *max* operator on j is lead by the score between the query term and the centroid, rather than the score between the query term and the residual. We argue that it is possible to compute the scores on the residuals only for a reduced set of document terms $\bar{J}_i$, where i identifies the index of the query term. In particular, $\bar{J}_i = \{j | q_i \cdot \bar{C}_j^T > th_r\}$, where th_r is a second threshold that determines whether the score with the centroid is sufficiently large. With the introduction of this new per-term filter, Equation 5 now becomes computing the max operator on the set of passages in $\bar{J}_i$, i.e.,

$$S_{q,P} = \sum_{i=1}^{n_q} \max_{j \in \bar{J}_i} (q_i \cdot \bar{C}^{T_j} + q_i \cdot r_{pq}^{T_j}). \tag{6}$$

In practice, we compute the residual scores only for those document terms whose centroid score is large enough. If $\bar{J}_i = \emptyset$, we compute $S_{q,P}$ as in Equation 5. We experimentally verify that this allows to save up to 30% in the late interaction with no performance degradation.

4. Experimental Evaluation

Experimental Settings. In this section, we compare our methodology with the state-of-the-art engine for multi-vector dense retrieval, namely PLAID [4]. Our experiments are conducted on the MS MARCO passages dataset [6] for in-domain evaluation and on LoTTE [3] for out-of-domain evaluation. Embeddings for MS MARCO are generated using the ColBERTv2 model, resulting in a dataset composed of about 600 million d-dimensional vectors, with $d = 128$. The implementation of Product Quantization utilizes the FAISS [7] library and is optimized using the JMPQ [8] technique. The experiments are carried out on an Intel Xeon Gold 5318Y CPU clocked at 2.10 GHz, equipped with the AVX512 instruction set, and executed with single-threading. The code is compiled using GCC 11.3.0 with -O3 compilation options on a Linux 5.15.0-72 machine.

Evaluation. Table 1 compares EMVB against PLAID on the MS MARCO dataset, in terms of memory requirements (num. of bytes per embedding), average query latency (in milliseconds), MRR@10, and Recall@100, and 1000. With $m = 16$, EMVB almost halves the per-vector memory load compared to PLAID, achieving up to $2.8\times$ faster processing with minimal impact on retrieval effectiveness. Doubling the number of sub-partitions per vector, i.e., $m = 32$, EMVB surpasses PLAID's performance in terms of MRR and Recall while maintaining the same memory footprint, achieving up to $2.5\times$ speedup.

Table 2 presents a comparison between EMVB and PLAID in the out-of-domain evaluation on the LoTTE dataset. Similar to PLAID [4], Success@5 and Success@100 are employed as retrieval quality metrics. On this dataset, EMVB exhibits slightly lower performance in terms of retrieval quality. It's worth noting that JMPQ [8] cannot be applied in the out-of-domain evaluation due to the absence of training queries. Instead, we utilize Optimized Product Quantization (OPQ) [9], which searches for an optimal rotation of the dataset vectors to mitigate the quality degradation associated with PQ. To address the retrieval quality loss, PQ is experimented with $m = 32$, as an increased number of partitions offers a better representation of the original vector. However, EMVB provides a substantial speedup of up to $2.9\times$ compared to PLAID in this out-of-domain

k	Method	Latency ($msec.$)	Bytes	MRR@10	R@100	R@1000
10	PLAID	131	36	39.4	-	-
	EMVB (m=16)	62 (2.1×)	20	39.4	-	-
	EMVB (m=32)	61 (2.1×)	36	39.7	-	-
100	PLAID	180	36	39.8	90.6	-
	EMVB (m=16)	68 (2.6×)	20	39.5	90.7	-
	EMVB (m=32)	80 (2.3×)	36	39.9	90.7	-
1000	PLAID	260	36	39.8	91.3	97.5
	EMVB (m=16)	93 (2.8×)	20	39.5	91.4	97.5
	EMVB (m=32)	104 (2.5×)	36	39.9	91.4	97.5

Table 1
Comparison between EMVB and PLAID in terms of average query latency, number of bytes per vector embeddings, MRR, and Recall on MS MARCO.

evaluation. This larger speedup, compared to MS MARCO, is attributed to the larger average document lengths in LoTTE. In this context, filtering non-relevant documents using our bit vector-based approach significantly impacts efficiency. It's noteworthy that for the out-of-domain evaluation, our pre-filtering method could be integrated into PLAID. This integration could maintain PLAID's accuracy while benefiting from EMVB's efficiency. Combinations of PLAID and EMVB are left for future exploration.

k	Method	Latency ($msec.$)	Bytes	Success@5	Success@100
10	PLAID	131	36	69.1	-
	EMVB (m=32)	82 (1.6×)	36	69.0	-
100	PLAID	202	36	69.4	89.9
	EMVB (m=32)	129 (1.6×)	36	69.0	89.9
1000	PLAID	411	36	69.6	90.5
	EMVB (m=32)	142 (2.9×)	36	69.0	90.1

Table 2
Comparison between EMVB and PLAID in terms of average query latency, number of bytes per vector embeddings, Success@5, and Success@100 on LoTTE.

Acknowledgments

This work was supported by the EU - NGEU, 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, by the PNRR ECS00000017 Tuscany Health Ecosystem Spoke 6 "Precision medicine & personalized healthcare", by the European Commission under the NextGeneration EU programme, by the Horizon Europe RIA "Extreme Food Risk Analytics" (EFRA), grant agreement n. 101093026, by the "Algorithms, Data Structures and Combinatorics for Machine Learning" (MIUR-PRIN 2017), and by the "Algorithmic Problems and Machine Learning" (MIUR-PRIN 2022).

References

[1] F. M. Nardini, C. Rulli, R. Venturini, Efficient multi-vector dense retrieval with bit vectors, in: Proceedings of the 46th European Conference on Information Retrieval (ECIR 2024), 2024.

[2] O. Khattab, M. Zaharia, Colbert: Efficient and effective passage search via contextualized late interaction over bert, in: Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval, 2020, pp. 39–48.

[3] K. Santhanam, O. Khattab, J. Saad-Falcon, C. Potts, M. Zaharia, Colbertv2: Effective and efficient retrieval via lightweight late interaction, in: Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2022.

[4] K. Santhanam, O. Khattab, C. Potts, M. Zaharia, Plaid: an efficient engine for late interaction retrieval, in: Proceedings of the 31st ACM International Conference on Information & Knowledge Management, 2022.

[5] H. Jegou, M. Douze, C. Schmid, Product quantization for nearest neighbor search, IEEE Transactions on Pattern Analysis and Machine Intelligence (2010).

[6] T. Nguyen, M. Rosenberg, X. Song, J. Gao, S. Tiwary, R. Majumder, L. Deng, Ms marco: A human-generated machine reading comprehension dataset (????).

[7] J. Johnson, M. Douze, H. Jegou, Billion-scale similarity search with gpus, IEEE Transactions on Big Data (2021).

[8] Y. Fang, J. Zhan, Y. Liu, J. Mao, M. Zhang, S. Ma, Joint optimization of multi-vector representation with product quantization, in: Natural Language Processing and Chinese Computing, 2022.

[9] T. Ge, K. He, Q. Ke, J. Sun, Optimized product quantization, IEEE Transactions on Pattern Analysis and Machine Intelligence (2013).