WARP: An Efficient Engine for Multi-Vector Retrieval

Jan Luca Scheerer*
lscheerer@ethz.ch
ETH Zurich
Switzerland

Matei Zaharia
matei@berkeley.edu
UC Berkeley
United States

Christopher Potts
cgpotts@stanford.edu
Stanford University
United States

Gustavo Alonso
alonso@inf.ethz.ch
ETH Zurich
Switzerland

Omar Khattab
okhattab@cs.stanford.edu
Stanford University
United States

arXiv:2501.17788v1 [cs.IR] 29 Jan 2025

Abstract

We study the efficiency of multi-vector retrieval methods like ColBERT and its recent variant XTR. We introduce WARP, a retrieval engine that drastically improves the efficiency of XTR-based ColBERT retrievers through three key innovations: (1) WARP$_{\text{SELECT}}$ for dynamic similarity imputation, (2) implicit decompression during retrieval, and (3) a two-stage reduction process for efficient scoring. Thanks also to highly-optimized C++ kernels and to the adoption of specialized inference runtimes, WARP can reduce end-to-end query latency relative to XTR's reference implementation by 41x, and thereby achieves a 3x speedup over the official ColBERTv2 PLAID engine, while preserving retrieval quality.

https://github.com/jlscheerer/xtr-warp

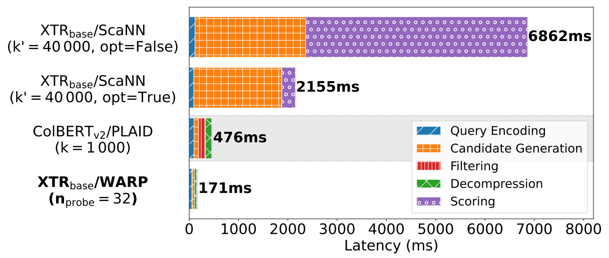


Figure 1: Single-threaded CPU latency breakdown of the unoptimized reference implementation from (1) XTR,[2](2) a variant of XTR that we optimized, (3) the official PLAID system, and (4) our proposed WARP on LoTTE Pooled.

Keywords

Dense Retrieval, Multi-Vector, Late Interaction, Efficiency

ACM Reference Format:
Jan Luca Scheerer, Matei Zaharia, Christopher Potts, Gustavo Alonso, and Omar Khattab. 2025. WARP: An Efficient Engine for Multi-Vector Retrieval. In *Preprint*. Preprint, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Over the past several years, information retrieval (IR) research has introduced new neural paradigms for search based on pretrained Transformers. Central among these, the late interaction paradigm proposed in ColBERT [8] departs the bottlenecks of conventional *single-vector* representations. It instead encodes queries and documents into *multi-vector* representations on top of which it is able to scale gracefully to search massive collections.

Since the original ColBERT was introduced, there has been substantial research in optimizing the latency of multi-vector retrieval models [2, 12, 13]. Perhaps most notably, PLAID [16] reduces late interaction search latency by 45x on a CPU compared to a vanilla ColBERTv2 [17] process, while continuing to deliver state-of-the-art retrieval quality. Orthogonally, Google DeepMind's ConteXtualized Token Retriever (XTR) [10] introduces a novel training objective that eliminates the need for a separate gathering stage and thereby significantly simplifies the subsequent scoring stage. While XTR lays extremely promising groundwork for more efficient multi-vector retrieval[1], we find that it relies naively on a general purpose vector similarity search library (ScANN) and combines that with native Python data structures and manual iteration, introducing substantial overhead.

The key insights from PLAID and XTR appear rather isolated. Whereas PLAID is concerned with aggressively and swiftly pruning away documents it finds unpromising, XTR tries to eliminate gathering complete document representations in the first place. We ask whether there are potential rich interactions between these two fundamentally distinct approaches to speeding up multi-vector search. To study this, we introduce a new engine for retrieval with XTR-based ColBERT models, called WARP, that combines techniques from ColBERTv2/PLAID with innovations tailored for the XTR architecture. Our contributions in WARP include: (1) the WARP$_{\text{SELECT}}$ method for imputing missing similarities, (2) a new method for implicit decompression of vectors during search, and (3) a novel two-stage reduction phase for efficient scoring.

Experimental evaluation shows that WARP achieves a 41x reduction in end-to-end latency compared to the XTR reference implementation on LoTTE Pooled, bringing query response times down from above 6 seconds to just 171 milliseconds in single-threaded execution, while also reducing index size by a factor of 2x–4x compared

*Work completed as a visiting student researcher at Stanford University.

[1]https://github.com/google-deepmind/xtr

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
Conference'17, July 2017, Washington, DC, USA
© 2025 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

to the ScaNN-based baseline. Furthermore, WARP demonstrates a 3x speedup over the state-of-the-art ColBERTv2/PLAID system, as illustrated in Figure 1.

After briefly reviewing prior work on efficient neural information retrieval in Section 2, we analyze the latency bottlenecks in the ColBERT and XTR retrieval frameworks in Section 3, identifying key areas for optimization within the XTR framework. These findings form the foundation for our work on WARP, which we introduce and describe in detail in Section 4. In Section 5, we evaluate WARP's end-to-end latency and scalability using the BEIR [18] and LoTTE [17] benchmarks. Finally, we compare our implementation to existing state-of-the-art engines.

2 Related Work

Dense retrieval models can be broadly categorized into single-vector and multi-vector approaches. Single-vector methods, exemplified by ANCE [19] and STAR/ADORE [20], encode a passage into a single dense vector [7]. While these techniques offer computational efficiency, their inherent limitation of representing complex documents with a single vector has been shown to constrain the model's ability to capture intricate information structures [8].

To address such limitations, ColBERT [8] introduces a multi-vector paradigm. In this approach, both queries and documents are independently encoded as multiple embeddings, allowing for a richer representation of document content and query intent. The multi-vector approach is further refined in ColBERTv2 [17], which improves supervision and incorporates residual compression to significantly reduce the space requirements associated with storing multiple vectors per indexed document. Building upon these innovations, PLAID [16] substantially accelerates ColBERTv2 by efficient pruning non-relevant passages using the residual representation and by employing optimized C++ kernels. EMVB [13] further optimizes PLAID's memory usage and *single-threaded* query latency using product quantization [6] and SIMD instructions.

Separately, COIL [2] incorporates insights from conventional retrieval systems [15] by constraining token interactions to lexical matches between queries and documents. SPLATE [1] translates the embeddings produced by ColBERTv2 style pipelines to a sparse vocabulary, allowing the candidate generation step to be performed using traditional *sparse* retrieval techniques. CITADEL [12] introduces conditional token interaction through dynamic lexical routing, selectively considering tokens for relevance estimation. While CITADEL significantly reduces GPU execution time, it falls short of PLAID's CPU performance at comparable retrieval quality.

The Conte**X**tualized **T**oken **R**etriever (XTR) [10], introduced by Lee et al. [10], represents a notable conceptual advancement in dense retrieval. XTR simplifies the scoring process and eliminates the gathering stage entirely, *theoretically* enhancing retrieval efficiency. However, its current end-to-end latency limits its application in production environments where query response time is critical or GPU resources are constrained.

3 Latency of Current Neural Retrievers

We start by analyzing two state-of-the-art multi-vector retrieval methods to identify their bottlenecks, providing the foundation for our work on WARP. We evaluate the latency of PLAID and XTR across various configurations and datasets: BEIR NFCorpus, LoTTE Lifestyle, and LoTTE Pooled. In XTR, token retrieval emerges as a fundamental bottleneck: the need to retrieve a large number of candidates from the ANN backend significantly impacts performance. PLAID, while generally far more efficient, faces challenges in its decompression stage. Query encoding emerges as a shared limitation for both engines, particularly on smaller datasets. These insights inform the design of WARP, which we introduce in Section 4.

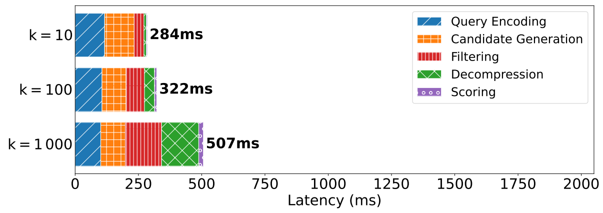


Figure 2: Breakdown of ColBERTv2/PLAID's avg. latency for varying k on LoTTE Pooled (Dev.)

3.1 ColBERTv2/PLAID

As shown in Figure 2, we evaluate PLAID's performance using its optimized implementation [4] and the ColBERTv2 checkpoint from Hugging Face [3]. We configure PLAID's hyperparameters similar to the original paper [16]. Consistent with prior work [13], we observe single threaded CPU exceeding 500ms on LoTTE Pooled. Furthermore, we find that the decompression stage remains rather constant for fixed k across all datasets, consuming approximately 150-200ms for $k = 1000$. Notably, for smaller datasets like NFCorpus and large k values, this stage constitutes a significant portion of the overall query latency. Thus, the decompression stage emerges as a critical bottleneck for small datasets. Query encoding contributes significantly to overall latency, particularly for smaller datasets and candidate generation consitutes a fixed cost based on the number of centroids. As anticipated, the filtering stage's execution time is proportional to the number of candidates, increasing for larger k values and bigger datasets. Interestingly, the scoring stage appears to have a negligible impact on ColBERTv2/PLAID's overall latency across all measurements.

3.2 XTR/ScaNN

To enable benchmarking of the XTR framework, we developed a Python library based on Google DeepMind's published code [11]. The library's code, along with scripts to reproduce the benchmarks, will be made available on GitHub. Unless otherwise specified, all benchmarks utilize the XTR BASE_EN transformer model for encoding. This model was published and is available on Hugging Face [9]. In accordance with the paper [10], we evaluate the implementation for k' = 1 000 and k' = 40 000.

As our measurements show, the scoring stage constitutes a significant bottleneck in the end-to-end latency of the XTR framework, particularly when dealing with large k' values, as seen in Figure 3. We argue that this performance bottleneck is largely attributed to an *unoptimized* implementation in the released code, which relies on native Python data structures and manual iteration, introducing

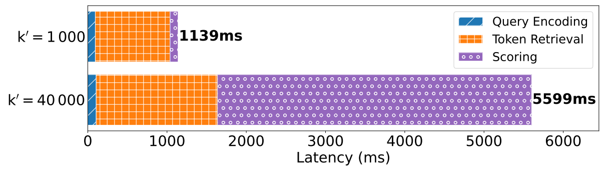


Figure 3: Breakdown of the Google DeepMind's reference implementation of XTR$_{\text{base}}$/ScaNN's avg. latency for varying k on LoTTE Pooled [17]

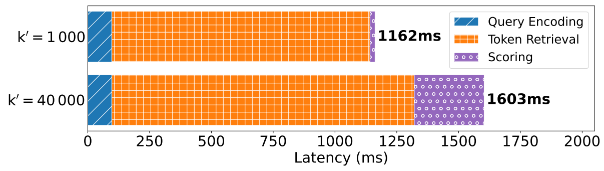


Figure 4: Figure 4 (c): Breakdown of XTR$_{\text{base}}$/ScaNN's avg. latency for varying k' on LoTTE Pooled [17]

substantial overhead, especially for large numbers of token embeddings. We refactored this implementation to leverage optimized data structures and vectorized operations. This helps us uncover hidden performance inefficiencies and establish a baseline for further optimization, with our improved implementation of XTR to be made publicly available.

We present the evaluation of our optimized implementation in Figure 4. Notably, the optimized implementation's end-to-end latency is significantly lower than that of the reference implementation ranging from an end-to-end 3.5x speed-up on LoTTE pooled to a 6.3x speed-up on LoTTE Lifestyle for $k = 1000$. This latency reduction is owed in large parts to a more efficient *scoring* implementation – 14x speed-up on LoTTE Pooled, see ??. In particular, this reveals token retrieval as the fundamental bottlenecks of the XTR framework. Furthermore, we notice that *query encoding* constitutes a large fraction of the overall end-to-end CPU latency on smaller datasets. While the optimized *scoring* stage consitutes a small fraction of the overall end-to-end latency, it is still slow in absolute terms – ranging from 33ms to 281ms for $k = 1000$ on BEIR NFCorpus and LoTTE Pooled, respectively.

4 WARP

WARP optimizes retrieval for the refined late interaction architecture introduced in XTR. Seeking to find the best of the XTR and PLAID worlds, **WARP introduces the novel WARP$_{\text{SELECT}}$ algorithm for candidate generation, which effectively avoids gathering token-level representations, and proposes an optimized two-stage reduction for faster scoring via a dedicated C++ kernel combined with implicit decompression.** WARP also uses specialized inference runtimes for faster query encoding.

As in XTR, queries and documents are encoded *independently* into embeddings at the token-level using a fine-tuned T5 transformer [14]. To scale to large datasets, document representations are computed in advance and constitute WARP's index. The similarity between a query q and document d is modeled using the

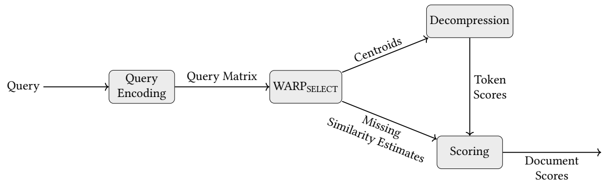


Figure 5: WARP Retrieval consisting of query encoding, WARP$_{\text{SELECT}}$, decompression, and scoring. Notably, centroid selection is combined with the computation of missing similarity estimates in WARP$_{\text{SELECT}}$.

XTR's adaptation of ColBERT's summation of MaxSim operations Equation (1):[3]

$$S_{d,q} = \sum_{i=1}^{n} \max_{1 \le j \le m} [\hat{\mathbf{A}}_{i,j} q_i^T d_j + (1 - \hat{\mathbf{A}}_{i,j}) m_i] \quad (1)$$

where q and d are the matrix representations of the query and passage embeddings respectively, m_i denotes the missing similarity estimate for q_i, and $\hat{\mathbf{A}}$ describes XTR's alignment matrix. In particular, $\hat{\mathbf{A}}_{i,j} = \mathbb{1}_{[j \in \text{top-}k'_j(\mathbf{d}_{i,j'})]}$ captures whether a document token embedding of a candidate passage was retrieved for a specific query token q_i as part of the token retrieval stage. We refer to [8, 10] for an intuition behind this choice of scoring function.

4.1 Index Construction

Akin to ColBERTv2 [17], WARP's compression strategy involves applying k-means clustering to the produced document token embeddings. As in ColBERTv2, we find that using a sample of all passages proportional to the square root of the collection size to generate this clustering performs well in practice. After having clustered the sample of passages, all token-level vectors are encoded and stored as quantized residual vectors to their nearest cluster centroid. Each dimension of the quantized residual vector is a b-bit encoding of the delta between the centroid and the original uncompressed vector. In particular, these deltas are stored as a sequence of $128 \cdot \frac{b}{8}$ 8-bit values, wherein 128 represents the transformers token embedding dimension. Typically, we set $b = 4$, i.e., compress each dimension of the residual into a single nibble, for an $8x$ compression[4]. In this case, each compressed 8-bit value stores 2 indices in the range $[0, 2^b]$. Instead of quantizing the residuals uniformly, WARP uses quantiles derived from the empirical distribution to determine bucket boundaries (`bucket_cutoffs`) and the corresponding representative values (`bucket_weights`). This process allows WARP to allocate more quantization levels to densely populated regions of the data distribution, thereby minimizing the overall quantization error for residual compression.

4.2 Retrieval

Extending PLAID, the retrieval process in the WARP engine is divided into four distinct steps: query encoding, candidate generation, decompression, and scoring. Figure 5 illustrates the retrieval process

[3]Note that we omit the normalization via $\frac{1}{Z}$, as we are only interested in the relative ranking between documents and the normalization constant is identical for any retrieved document.

[4]As compared to an uncompressed 32-bit floating point number per dimension.

in WARP. The process starts when the query text is encoded into q, a (query_maxlen, 128)-dimensional tensor, using the Transformer model.[5] The most similar n_{probe} centroids are identified for each of the query_maxlen query token embeddings.

Subsequently, WARP identifies all document token embeddings belonging to the clusters of the selected centroids and computes their *individual* relevance score. Computing this score involves decompressing residuals of these identified document token embeddings and calculating the cosine similarity with the relevant query token embedding. Finally, WARP (implicitly) constructs a $n_{\text{candidates}} \times$ query_maxlen score matrix S,[6] wherein each entry S_{d_i,q_j} contains the maximum retrieved score for the i-th candidate passage d_i and the j-th query token embedding q_j:

$$\max_{1\leq j\leq m} \hat{\mathrm{A}}_{i,j} q_i^T d_j$$

Matrix entries not populated during token retrieval, i.e., $\hat{\mathrm{A}}_{i,j} = 0$, are each imputed with a missing similarity estimate, as postulated in the XTR framework. To compute the relevance score of a document d_i, the cummulative score over all query tokens is computed: $\sum_j S_{d_i,q_j}$. To produce the ordered set of passages, the set of scores sorted and the top k highest scoring passages are returned.

4.3 WARP$_{\text{SELECT}}$

In contrast to ColBERT, which populates the entire score matrix for the items retrieved, XTR only populates the score matrix with scores computed as part of the token retrieval stage. To account for the contribution of any missing tokens, XTR relies on *missing similarity imputation*, in which they set any missing similarity of the query token for a specific document as the lowest score obtained as part of the gathering stage. The authors argue that this approach is justified as it constitutes a *natural* upper bound for the true relevance score. In the case of WARP, this bound is no longer guaranteed to hold.[7]

Instead, WARP defines a novel strategy for candidate generation based on cumulative cluster sizes, WARP$_{\text{SELECT}}$. Given the query embedding matrix q and the list of centroids C (Section 4.1), WARP computes the token-level query-centroid relevance scores. As both the query embedding vectors and the set of centroids are normalized, the cosine similarity scores $S_{c,q}$ can be computed efficiently as a matrix multiplication:

$$S_{c,q} = C \cdot q^T$$

Once these relevance scores have been computed WARP identifies the n_{probe} centroids with the largest similarity scores for decompression, as part of candidate generation.

Using these query–centroid similarity scores, WARP$_{\text{SELECT}}$ folds the estimation of missing similarity scores into candidate generation. Specifically, for each query token q_i, it sets m_i from Equation (1) as the first element in the sorted list of centroid scores for which the cumulative cluster size exceeds a threshold t'. This method is particularly attractive as all the centroid scores have already been computed and sufficiently sorted as part of candidate generation, so the cost of computing missing similarity imputation with this method is negligible.

We find that t' is easy to configure (Section 4.6) without compromising the retrieval quality or efficiency. Unlike XTR, the missing similarity estimate of WARP is inherently tied to the number of retrieved tokens. Intuitively, increasing k' may only help refine the missing similarity estimate, but not significantly increase the density of the score matrix.[8]

4.4 Decompression

The input for the decompression phase is the set of n_{probe} centroid indices for each of the `query_maxlen` query tokens. Its goal is to calculate relevance scores between each query token and the embeddings within the identified clusters. For a query token q_i, let $c_{i,j}$, where $j \in [n_{\text{probe}}]$, be the set of centroid indices identified during candidate generation. Let $r_{i,j,k}$ be the set of residuals associated with cluster $c_{i,j}$. The decompression step computes:

$$s_{i,j,k} = \text{decompress}(C[c_{i,j}], r_{i,j,k}) \times q_i^T \;\forall\; i,j,k \tag{2}$$

The decompress function converts residuals from their compact form from ColBERTv2 and PLAID into uncompressed vectors. Each residual $r_{i,j,k}$ is composed of 128 indices, each b bits wide. These indices reference values in the bucket weights vector $\omega \in \mathbb{R}^{2^b}$ and are used to offset the centroid $C[c_{i,j}]$. The decompress function is defined as:

$$\text{decompress}(C[c_{i,j}], r_{i,j,k}) = C[c_{i,j}] + \sum_{d=1}^{128} \boldsymbol{e}_d \cdot \omega[(r_{i,j,k})_d] \tag{3}$$

Here, $\boldsymbol{e}_d$ is the unit vector for dimension d, and $\omega[(r_{i,j,k})_d]$ is the weight value at index $(r_{i,j,k})_d$ for dimension d. In other words, the indices are used to look up specific entries in ω for each dimension independently, adjusting the centroid accordingly.

Instead of *explicitly* decompressing residuals as in PLAID, WARP leverages the observation that the scoring function decomposes between centroids and residuals. As a result, WARP reuses the query-centroid relevance scores $S_{c,q}$, computed as part of candidate generation. That is, observe that:

$$\begin{aligned} s_{i,j,k} &= \text{decompress}(C[c_{i,j}], r_{i,j,k}) \times q_i^T \\ &= (C[c_{i,j}] \times q_i^T) + (\sum_{d=1}^{128} \omega[(r_{i,j,m,k})_d] q_{i,d}) \end{aligned} \tag{4}$$

To accelerate decompression, WARP computes $v = \hat{q} \times \hat{\omega}$, wherein $\hat{q} \in \mathbb{R}^{\texttt{query\_maxlen} \times 128 \times 1}$ represents the query matrix that has been *unsqueezed* along the last dimension, and $\hat{\omega} \in \mathbb{R}^{1 \times 2^b}$ denotes the vector of bucket weights that has been *unsqueezed* along the first dimension. With these definitions, WARP can decompress and score candidate tokens via:

$$\begin{aligned} s_{i,j,k} &= S_{c_j,q_i} + \sum_{d=1}^{128} (\omega \cdot q_{i,d})[(r_{i,jk})_d] \\ &= S_{c_j,q_i} + \sum_{i=1}^{128} v_{i,d}[(r_{i,j,k})_d] \end{aligned} \tag{5}$$

[5]We set query_maxlen = 32 in accordance with the XTR paper.
[6]Note that our optimized implementation does not physically construct this matrix.
[6]For XTR baselines, 'candidate generation' refers to the token retrieval stage.
[7]Strictly speaking, the bound is also approximate in XTR's case, as ScaNN does not return the exact nearest neighbors in general.
[8]This is because tokens retrieved with a larger k' are often from new documents and, therefore, do not refine the scores of already retrieved ones.

Note that candidate scoring can now be implemented as a simple *selective sum*. As the bucket weights are shared among centroids and the query-centroid relevance scores have already been computed during candidate generation, WARP can decompress and score arbitrarily many clusters using $O(1)$ multiplications. **This refined scoring function is far more efficient then the one outlined in PLAID,[9] as it never computes the decompressed embeddings explicitly and instead directly emits the resulting candidate scores.** We provide an efficient implementation of the selective sum of Equation (4) and realize unpacking of the residual representation using low-complexity bitwise operations as part of WARP's C++ kernel for decompression.

4.5 Scoring

At the end of the decompression phase, we have query_maxlen $\times$ n_{probe} strides of decompressed candidate document *token-level scores* and their corresponding document identifiers. Scoring combines these scores with the missing similarity estimates, computed during candidate generation, to produce *document-level scores*. This process corresponds to constructing the score matrix and taking the row-wise sum.

Explicitly constructing the score matrix, as in the the reference XTR implementation, introduces a significant bottleneck, particularly for large values of n_{probe}. To address this, WARP efficiently aggregates token-level scores using a two-stage reduction process:

- **Token-level reduction** For each query token, reduce the corresponding set of n_{probe} strides using the max operator. This step *implicitly* fills the score matrix with the maximum per-token score for each document. As a single cluster can contain multiple document token embeddings originating from the same document, WARP performs *inner-cluster* max-reduction directly during the decompression phase.
- **Document-level reduction** Reduce the resulting strides into document-level scores using a sum aggregation. It is essential to handle missing values properly at this stage – any missing per-token score must be replaced by the corresponding missing similarity estimate, ensuring compliance with the XTR scoring function described in Equation (1). This reduction step corresponds to the row-wise summation of the score matrix.

After performing both reduction phases, the final stride contains the document-level scores and the corresponding identifiers for all candidate documents. To retrieve the result set, we perform heap select to obtain the top-k documents, similar to its use in the candidate generation phase. Formally, we consider a stride S to be an list of key-value pairs:

$$S = \{(k_i, v_i)\};\ \mathrm{K}(S) = \{k_i \mid (k_i, v_i) \in S\};\ \mathrm{V}(S) = \{v_i \mid (k_i, v_i) \in S\}$$

Thus, strides implicitly define a partial function $f_S : K \rightharpoonup V(S)$:

$$f_S(k) = \begin{cases} v_i & \text{if } \exists v_i.\ (k, v_i) \in S \\ \perp & \text{otherwise} \end{cases}$$

We define a reduction as a combination of two strides S_1 and S_2 using a binary function r into a single stride by applying r to values of matching keys:

$$\text{reduce}(r, S_1, S_2) = \{(k, r(f_{S_1}(k), f_{S_2}(k))) \mid k \in K(S_1) \cup K(S_2)\}$$

With these definitions, token-level reduction can be written as:

$$r_{\text{tok}}(v_1, v_2) = \begin{cases} \max(v_1, v_2) & \text{if } v_1 \neq \perp \wedge v_2 \neq \perp \\ v_1 & \text{if } v_1 \neq \perp \wedge v_2 = \perp \\ v_2 & \text{otherwise} \end{cases} \tag{6}$$

Defining the document-level reduction is slightly more complex as it involves incorporating the *corresponding* missing similarity estimates m. After token-level reduction each of the query_maxlen strides $S_1, \ldots, S_{\text{query\_maxlen}}$ *covers* scores for a single query token q_i. We set $S_{i,i} = S_i$ and define:

$$S_{i,j} = \text{reduce}(r_{\text{doc},(i,k,j)}, S_{i,k}, S_{k+1,j}) \tag{7}$$

for any choice of $i \leq k < j$, wherein $r_{\text{doc},(i,k,j)}$ merges two successive, non-overlapping strides $S_{i,k}$ and $S_{k+1,j}$. The resulting stride, $S_{i,j}$, now covers scores for query tokens $q_i, \ldots, q_j$. Defining $r_{\text{doc},(i,k,j)}$ is relatively straightforward:

$$r_{\text{doc},(i,k,j)}(v_1, v_2) = \begin{cases} v_1 + v_2 & \text{if } v_1 \neq \perp \wedge v_2 \neq \perp \\ v_1 + (\sum_{t=k+1}^{j} m_t) & \text{if } v_1 \neq \perp \wedge v_2 = \perp \\ (\sum_{t=i}^{k} m_t) + v_2 & \text{otherwise} \end{cases} \tag{8}$$

It is easy to verify that $S_{i,j}$ is well-defined, i.e., independent of the choice of k. The result of document-level reduction is $S_{1,\text{query\_maxlen}}$ and can be obtained by recursively applying Equation (7) to strides of increasing size.

WARP's two-stage reduction process, along with the final sorting step, is illustrated in Figure 6. In the token-level reduction stage, strides are merged by selecting the maximum value for matching keys. In the document-level reduction stage, values for matching keys are summed, with missing values being substituted by the corresponding missing similarity estimates.

In our implementation, we conceptually construct a binary tree of the required merges and alternate between two scratch buffers to avoid additional memory allocations. We realize Equation (8) using a prefix sum, which eliminates the need to compute the sum explicitly.

4.6 Hyperparameters

In this section, we aim to analyze the effects of WARP's three primary hyperparameters, namely:

- n_{probe} – the #clusters to decompress per query token
- t' – the threshold on the cluster size used for WARP$_{\text{SELECT}}$
- b – the number of bits per dimension of a residual vector

To study the effects of n_{probe} and t', we analyze the normalized Recall@100[10] as a function of t' for $n_{\text{probe}} \in \{1, 2, 4, 8, 16, 32, 64\}$ across four development datasets of increasing size: BEIR NFCorpus, BEIR Quora, LoTTE Lifestyle, and LoTTE Pooled. For further details on the datasets, please refer to Table 1. Figure 7 visualizes the results of our analysis. We observe a consistent pattern across

[9]PLAID cannot adopt this approach directly, as it *normalizes* the vectors after decompression. Empirically, we find that that this normalization step has limited effect on the final embeddings as the residuals are already normalized prior to quantization.

[10]The normalized Recall@k is calculated by dividing Recall@k by the dataset's maximum, effectively scaling values between 0 and 1 to ensure comparability across datasets.

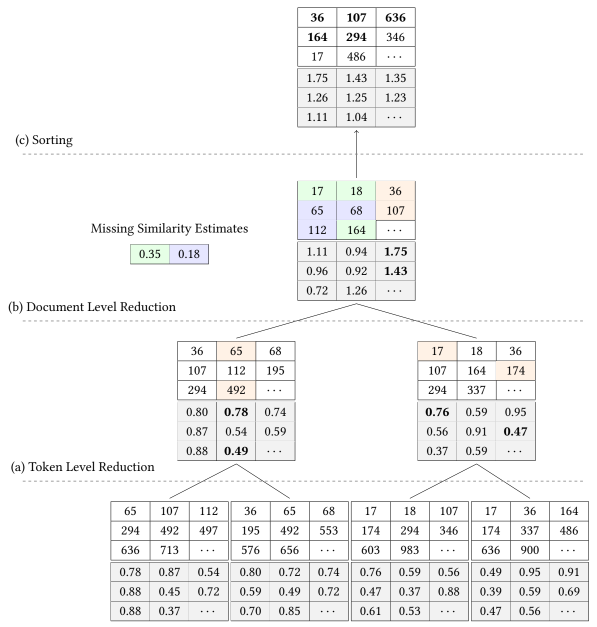
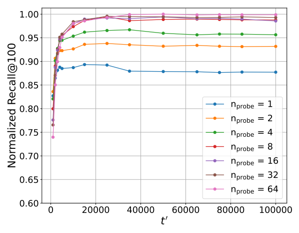
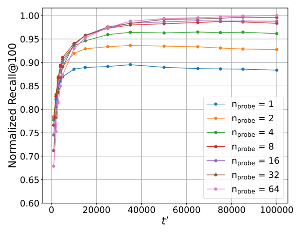

(c) Sorting
Missing Similarity Estimates
(b) Document Level Reduction
(a) Token Level Reduction

Figure 6: WARP's scoring phase: (a) In token-level reduction, strides are max-reduced. (b) In document-level reduction, values are summed, accounting for missing similarity estimates. (c) Scores are sorted, yielding the top-k results.

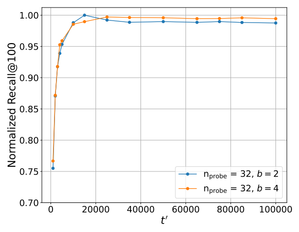


(a) LoTTE Science (Dev Set)

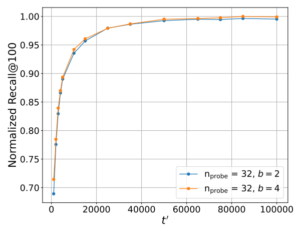


(b) LoTTE Pooled (Dev Set)

Figure 7: nRecall@100 as a function of t' and n_{probe}



(a) LoTTE Science (Dev Set)



(b) LoTTE Pooled (Dev Set)

Figure 8: nRecall@100 as a function of t' and b

all evaluated datasets, namely substantial improvements as n_{probe} increases from 1 to 16 (i.e., 1, 2, 4, 8, 16), followed by only marginal gains in Recall@100 beyond that. A notable exception is BEIR NFCorpus, where we still observe significant improvement when increasing from $n_{\text{probe}} = 16$ to $n_{\text{probe}} = 32$. We hypothesize that this is due to the small number of embeddings per cluster in NFCorpus, limiting the number of scores available for aggregation. Consequently, we conclude that setting $n_{\text{probe}} = 32$ strikes a good balance between end-to-end latency and retrieval quality.

In general, we find that WARP is highly robust to variations in t'. However, smaller datasets, such as NFCorpus, appear to benefit from a smaller t', while larger datasets perform better with a larger t'. Empirically, we find that setting t' proportional to the square root of the dataset size consistently yields strong results across all datasets. Moreover, increasing t' beyond a certain point no longer improves Recall, leading us to bound t' by a maximum value, t'_{max}.

Next, we aim to quantify the effect of b on the retrieval quality of WARP, as shown in Figure 8. To do this, we compute the nRecall@k for $n_{\text{probe}} = 32$ and $k \in \{10, 100\}$ using two datasets: LoTTE Science and LoTTE Pooled. Our results show a significant improvement in retrieval performance when increasing b from 2 to 4, particularly for smaller values of k. For larger values of k, the difference in performance diminishes, particularly for the LoTTE Pooled dataset.

5 Evaluation

We now evaluate WARP on six datasets from BEIR [18] and six datasets from LoTTE [17] listed in Table 1. We use servers with

Dataset		Dev		Test	
		#Queries	#Passages	#Queries	#Passages
BeIR [18]	NFCORPUS	324	3.6K	323	3.6K
	SciFact	–	–	300	5.2K
	SCIDOCS	–	–	1,000	25.7K
	Quora	5,000	522.9K	10,000	522.9K
	FiQA-2018	500	57.6K	648	57.6K
	Touché-2020	–	–	49	382.5K
LoTTE [17]	Lifestyle	417	268.9K	661	119.5K
	Recreation	563	263.0K	924	167.0K
	Writing	497	277.1K	1,071	200.0K
	Technology	916	1.3M	596	638.5K
	Science	538	343.6K	617	1.7M
	Pooled	2,931	2.4M	3,869	2.8M

Table 1: Datasets used for evaluating XTR$_{\text{base}}$/WARP performance. The evaluation includes 6 datasets from BEIR [18] and 6 from LoTTE [17].

28 `Intel Xeon Gold 6132 @ 2.6 GHz` CPU cores[11] and 500 GB RAM. The servers have two NUMA sockets with roughly 92 ns intra-socket memory latency, 142 ns inter-socket memory latency, 72 GBps intra-socket memory bandwidth, and 33 GBps inter-socket memory bandwidth.

When measuring latency for end-to-end results, we compute the average latency of all queries Table 1 and report the minimum average latency across three trials. For other results, we describe the specific measurement procedure in the relevant section. We measure latency on an otherwise idle machine. As XTR's token

[11]Each core has 2 threads for a total of 56 threads.

	Lifestyle	Recreation	Writing	Technology	Science	Pooled	Avg.
BM25	63.8	56.5	60.3	41.8	32.7	48.3	50.6
ColBERT	80.2	68.5	74.7	61.9	53.6	67.3	67.7
GTR$_{base}$	82.0	65.7	74.1	58.1	49.8	65.0	65.8
XTR/ScaNN	**83.5** (333.6)	**69.6** (400.2)	78.0 (378.0)	63.9 (742.5)	55.3 (1827.6)	68.4 (2156.3)	69.8
WARP	**83.5** (73.1)	69.5 (72.4)	**78.6** (73.6)	**64.6** (96.4)	**56.1** (156.4)	**69.3** (171.3)	**70.3**
Splade$_{v2}$ ♣◊	82.3	69.0	77.1	62.4	55.4	68.9	69.2
ColBERT$_{v2}$ ♣◊	84.7	72.3	80.1	66.1	56.7	71.6	71.9
GTR$_{xxl}$	87.4	78.0	83.9	69.5	60.0	76.0	75.8
XTR$_{xxl}$	89.1	79.3	83.3	73.7	60.8	77.3	77.3

♣: cross-encoder distillation ◊: model-based hard negatives

Table 2: Success@5 on LoTTE. Numbers in parentheses show average latency (milliseconds), with the final column displaying the average score across the datasets. Both XTR/ScaNN and WARP use the model XTR$_{base}$. XTR/ScaNN uses $k' = 40000$ and WARP uses $n_{nprobe} = 32$.

	NFCorpus	SciFact	SCIDOCS	FiQA-2018	Touché-2020	Quora	Avg.
BM25	25.0	90.8	35.6	53.9	53.8	97.3	59.4
ColBERT	25.4	87.8	34.4	60.3	43.9	98.9	58.5
GTR$_{base}$	27.5	87.2	34.0	67.0	44.3	99.6	59.9
T5-ColBERT$_{base}$	27.6	91.3	34.2	63.0	49.9	97.9	60.7
XTR/ScaNN	**28.0** (158.1)	90.9 (309.7)	34.3 (297.3)	62.0 (338.2)	50.3 (560.2)	98.9 (411.2)	60.7
WARP	27.9 (58.0)	**92.8** (64.3)	**36.8** (66.1)	**62.3** (70.7)	**51.5** (94.8)	**99.0** (67.6)	**61.7**
GTR$_{xxl}$	30.0	90.0	36.6	78.0	46.6	99.7	63.5
T5-ColBERT$_{xxl}$	29.0	94.6	38.5	72.5	50.1	99.1	64.0
XTR$_{xxl}$	30.7	95.0	39.4	73.0	52.7	99.3	65.0

Table 3: Recall@100 on BEIR. Numbers in parentheses show average latency (milliseconds), with the final column displaying the average score across the datasets.

retrieval stage does not benefit from GPU acceleration due to it's use of ScaNN [5], specifically designed for single-threaded[12] use on x86 processors with AVX2 support. Therefore, we perform CPU-only measurements and restrict the usage to a single thread unless otherwise stated.

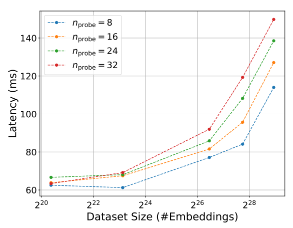
5.1 End-to-End Results

Table 2 presents results on LoTTE. Our WARP method over the XTR model outperforms the optimized XTR$_{base}$/ScaNN implementation in terms of Success@5, while significantly reducing end-to-end latency, with speedups ranging from 4.6x on LoTTE Lifestyle to 12.8x on LoTTE Pooled. We observe a similar trend with the evaluation of nDCG@10 on the six BEIR [18] datasets, as shown in ??. XTR$_{base}$/WARP achieves speedups of 2.7x-6x over XTR$_{base}$/ScaNN with a slight gain in nDCG@10. Likewise, we find improvements of Recall@100 on BEIR with substantial gains in end-to-end latency, but we omit them for the sake of space.

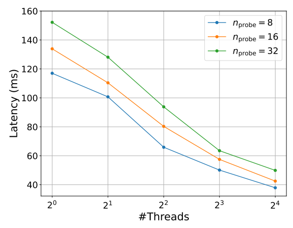
5.2 Scalability

We now assess WARP's scalability in relation to both dataset size and the degree of parallelism. To study the effect of the dataset size on WARP's performance, we evaluate its latency across development datasets of varying sizes: BEIR NFCorpus, BEIR Quora, LoTTE Science, LoTTE Technology, and LoTTE Pooled (Table 1). Figure 9a plots the recorded latency of different configurations versus the size of the dataset, measured in the number of document token embeddings. Our results confirm, like in PLAID, that WARP's latency scales in general with the square root of the dataset size—this is intuitive, as the number of clusters is by design proportional to the square root of the dataset size. Figure 9b illustrates WARP's performance on the LoTTE Pooled development set, showing how the number of CPU threads impacts performance for different values of n_{probe}. Our results indicate that WARP effectively parallelizes across multiple threads, achieving a speedup of 3.1x for $n_{probe} = 32$ with 16 threads.

[12]As of the recently released version 1.3.0, ScaNN supports multi-threaded search via the `search_batched_parallel` function.



(a) End-to-end latency vs dataset size (measured in #embeddings)



(b) End-to-end latency for varying n_{probe}

Figure 9: WARP's scaling behaviour with respect to dataset size and the number of available CPU threads

5.3 Memory footprint

Dataset		# Tokens	XTR Index Size (GiB)				
			BruteForce	FAISS	ScaNN	WARP$_{(b=2)}$	WARP$_{(b=4)}$
BeIR [18]	NFCORPUS	1.35M	0.65	0.06	0.18	0.06	0.10
	SciFact	1.87M	0.91	0.08	0.25	0.07	0.13
	SCIDOCS	6.27M	3.04	0.28	0.82	0.24	0.43
	Quora	9.12M	4.43	0.43	1.21	0.35	0.62
	FiQA-2018	10.23M	4.95	0.46	1.34	0.38	0.69
	Touché-2020	92.64M	45.01	4.28	12.22	3.40	6.16
LoTTE [17]	Lifestyle	23.71M	11.51	1.08	3.12	0.88	1.59
	Recreation	30.04M	14.59	1.38	3.96	1.11	2.01
	Writing	32.21M	15.64	1.48	4.25	1.19	2.15
	Technology	131.92M	64.12	6.13	17.44	4.83	8.77
	Science	442.15M	214.93	20.57	58.46	16.07	29.28
	Pooled	660.04M	320.88	30.74	87.30	23.88	43.59
Total	–	1.44B	700.66	66.98	190.52	**52.48**	95.51

Table 4: Comparison of index sizes for the datasets. Note that PLAID's usage is memory usage is effectively identical to WARP's, only slightly larger.

Thanks to the adoption of a ColBERTv2- and PLAID-like approach for compression, WARP's advantage over XTR extends also to a reduction in index size for XTR-based methods, which decreases memory requirements and, thus, broadens deployment options. Table 4 compares index sizes across all evaluated test datasets. WARP$_{(b=4)}$ demonstrates a substantially smaller index size compared to both the BruteForce and ScaNN variants, providing a 7.3x and 2x reduction in memory footprint, respectively. While indexes generated by the FAISS implementation are marginally smaller, this comes at the cost of substantially reduced quality and latency. Notably, WARP$_{(b=2)}$ outperforms the FAISS implementation in terms of quality with an even smaller index size.[13]

6 Conclusion

We introduce WARP, a highly optimized engine for multi-vector retrieval based on ColBERTv2 PLAID and the XTR framework. WARP overcomes inefficiencies of existing engines by: (1) the acceleration of query encoding using specialized inference runtimes, reducing inference latency by 2x-3x, (2) WARP$_{\text{SELECT}}$, which dynamically adjusts to dataset characteristics while simultaneously decreasing computational overhead, and (3) an optimized two-stage reduction via a dedicated C++ kernel combined with implicit decompression. These optimizations culminate in substantial performance gains, including a 41x speedup over XTR on LoTTE Pooled, reducing latency from above 6s to just 171ms in single-threaded execution, and a 3x reduction in latency compared to ColBERTv2/PLAID, without negatively impacting retrieval quality. Beyond single-threaded performance, WARP shows significant speedup with increased thread count, and its reduced memory footprint enables deployment on resource-constrained devices.

References

[1] Thibault Formal, Stéphane Clinchant, Hervé Déjean, and Carlos Lassance. 2024. Splate: sparse late interaction retrieval. (2024). arXiv: 2404.13950 [cs.IR].

[2] Luyu Gao, Zhuyun Dai, and Jamie Callan. 2021. Coil: revisit exact lexical match in information retrieval with contextualized inverted list. (2021). https://arxiv.org/abs/2104.07186 arXiv: 2104.07186 [cs.IR].

[3] Stanford Future Data Systems Research Group. 2024. colbert-ir/colbertv2.0. https://huggingface.co/colbert-ir/colbertv2.0. (2024).

[4] Stanford Future Data Systems Research Group. 2024. ColBERTv2/PLAID (Code). https://github.com/stanford-futuredata/ColBERT. (2024).

[5] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating large-scale inference with anisotropic vector quantization. (2020). https://arxiv.org/abs/1908.10396 arXiv: 1908.10396 [cs.LG].

[6] Herve Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33, 1, 117–128. doi: 10.1109/TPAMI.2010.57.

[7] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, (Eds.) Association for Computational Linguistics, Online, (Nov. 2020), 6769–6781. doi: 10.18653/v1/2020.emnlp-main.550.

[8] Omar Khattab and Matei Zaharia. 2020. Colbert: efficient and effective passage search via contextualized late interaction over BERT. *CoRR*, abs/2004.12832. https://arxiv.org/abs/2004.12832 arXiv: 2004.12832.

[9] Jinhyuk Lee, Zhuyun Dai, Sai Meher Karthik Duddu, Tao Lei, Iftekhar Naim, Ming-Wei Chang, and Vincent Y. Zhao. 2024. google/xtr-base-en. https://huggingface.co/google/xtr-base-en. (2024).

[10] Jinhyuk Lee, Zhuyun Dai, Sai Meher Karthik Duddu, Tao Lei, Iftekhar Naim, Ming-Wei Chang, and Vincent Y. Zhao. 2024. Rethinking the role of token retrieval in multi-vector retrieval. (2024). arXiv: 2304.01982 [cs.CL].

[11] Jinhyuk Lee, Zhuyun Dai, Sai Meher Karthik Duddu, Tao Lei, Iftekhar Naim, Ming-Wei Chang, and Vincent Y. Zhao. 2024. XTR: Rethinking the Role of Token Retrieval in Multi-Vector Retrieval (Code). https://github.com/google-deepmind/xtr. (2024).

[12] Minghan Li, Sheng-Chieh Lin, Barlas Oguz, Asish Ghoshal, Jimmy Lin, Yashar Mehdad, Wen-tau Yih, and Xilun Chen. 2022. Citadel: conditional token interaction via dynamic lexical routing for efficient and effective multi-vector retrieval. (2022). https://arxiv.org/abs/2211.10411 arXiv: 2211.10411 [cs.IR].

[13] Franco Maria Nardini, Cosimo Rulli, and Rossano Venturini. 2024. Efficient multi-vector dense retrieval using bit vectors. (2024). arXiv: 2404.02805 [cs.IR].

[14] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2023. Exploring the limits of transfer learning with a unified text-to-text transformer. (2023). https://arxiv.org/abs/1910.10683 arXiv: 1910.10683 [cs.LG].

[15] Stephen Robertson and Hugo Zaragoza. 2009. The probabilistic relevance framework: bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3, 4, 333–389. doi: 10.1561/1500000019.

[16] Keshav Santhanam, Omar Khattab, Christopher Potts, and Matei Zaharia. 2022. Plaid: an efficient engine for late interaction retrieval. (2022). arXiv: 2205.09707 [cs.IR].

[17] Keshav Santhanam, Omar Khattab, Jon Saad-Falcon, Christopher Potts, and Matei Zaharia. 2021. Colbertv2: effective and efficient retrieval via lightweight late interaction. *CoRR*, abs/2112.01488. https://arxiv.org/abs/2112.01488 arXiv: 2112.01488.

[18] Nandan Thakur, Nils Reimers, Andreas Rücklé, Abhishek Srivastava, and Iryna Gurevych. 2021. Beir: a heterogenous benchmark for zero-shot evaluation of information retrieval models. (2021). https://arxiv.org/abs/2104.08663 arXiv: 2104.08663 [cs.IR].

[19] Lee Xiong, Chenyan Xiong, Ye Li, Kwok-Fung Tang, Jialin Liu, Paul Bennett, Junaid Ahmed, and Arnold Overwijk. 2020. Approximate nearest neighbor negative contrastive learning for dense text retrieval. (2020). https://arxiv.org/abs/2007.00808 arXiv: 2007.00808 [cs.IR].

[20] Jingtao Zhan, Jiaxin Mao, Yiqun Liu, Jiafeng Guo, Min Zhang, and Shaoping Ma. 2021. Optimizing dense retrieval model training with hard negatives. (2021). https://arxiv.org/abs/2104.08051 arXiv: 2104.08051 [cs.IR].

[13] Additionally, WARP reduces memory requirements compared to PLAID as it no longer requires storing a mapping from document ID to centroids/embeddings.

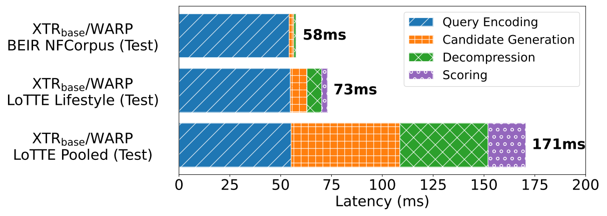


Figure 10: Breakdown of XTR$_{base}$/WARP's avg. single-threaded latency for n_{probe} = 32 on the BEIR NFCorpus, LoTTE Lifestyle, and LoTTE Pooled datasets.

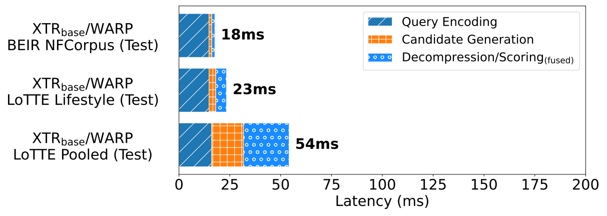


Figure 11: Breakdown of XTR$_{base}$/WARP's avg. latency for n_{probe} = 32 and $n_{threads}$ = 16 on the BEIR NFCorpus, LoTTE Lifestyle, and LoTTE Pooled datasets

A Additional Results

A.1 Latency Breakdowns

Next, we provide a more detailed breakdown of XTR$_{base}$/WARP's performance on three datasets of varying sizes: BEIR NFCorpus [18], LoTTE Lifestyle [17], and LoTTE Pooled [17]. Figure 10 illustrates the latency breakdown across four key stages: query encoding, candidate generation, decompression, and scoring. For the smallest dataset, BEIR NFCorpus, the total latency is 58ms, with query encoding dominating the process. Moving to the larger LoTTE Lifestyle dataset, the total latency increases to 73ms. Notably, on this dataset with over 100K passages, WARP's entire retrieval pipeline – comprising candidate generation, decompression, and scoring – constitutes only about 25% of the end-to-end latency, with the remaining time spent on query encoding. Even for the largest dataset, LoTTE Pooled, where the total latency reaches 171ms, we observe that query encoding still consumes the majority of the processing time. While the other stages become more pronounced, query encoding remains the single most time-consuming stage of the retrieval process. Without the use of specialized inference runtimes, query encoding accounts for approximately half of the execution time, thus presenting the primary bottleneck for end-to-end retrieval using WARP.

A key advantage of WARP over the reference implementation is its ability to leverage multi-threading, thereby significantly improving performance. Figure 11 illustrates the end-to-end latency breakdown for WARP using 16 threads. The decompression and scoring stages are fused in multi-threaded contexts. WARP demonstrates great scalability, achieving substantial latency reduction across all stages. In the 16-thread configuration, it notably surpasses the GPU-based implementation of PLAID on the LoTTE Pooled dataset.

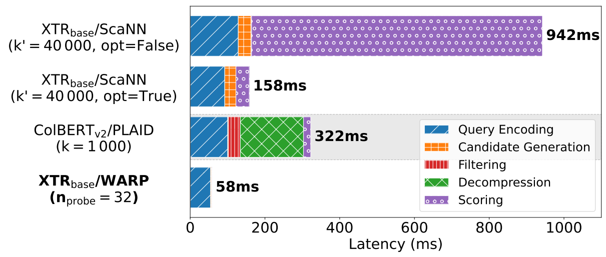


Figure 12: Latency breakdown of the unoptimized reference implementation, optimized variant, ColBERTv2/PLAID, and XTR$_{base}$/WARP on BEIR NFCorpus Test

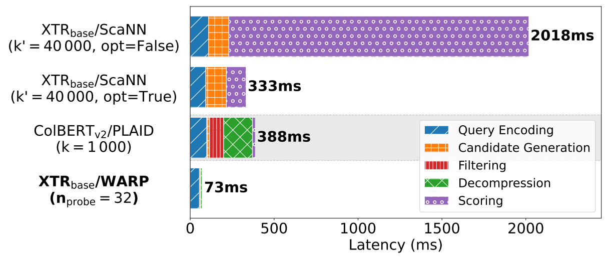


Figure 13: Latency breakdown of the unoptimized reference implementation, optimized variant, ColBERTv2/PLAID, and XTR$_{base}$/WARP on LoTTE Lifestyle Test

A.2 Performance Comparisons

Similar to Figure 1, we analyze the performance of WARP and contrast it with the performance of the baseline on the BEIR NFCorpus (Figure 12) and LoTTE Lifestyle (Figure 13) datasets. We find that WARP's single-threaded end-to-end latency is dominated by query encoding on BEIR NFCorpus and LoTTE Lifestyle, whereas the baselines introduce significant overhead via their retrieval pipeline.

A.3 Evaluation of ColBERTv2/WARP

To assess WARP's ability to generalize beyond the XTR model, we conduct experiments using ColBERTv2 in place of XTR$_{base}$ for query encoding. The results, presented in Table 5, show that WARP performs competitively with PLAID, despite not being specifically designed for retrieval with ColBERTv2. This suggests that WARP's approach may generalize effectively to retrieval models other than XTR. We leave more detailed analysis for future work.

	NFCorpus	SciFact	SCIDOCS	FiQA-2018	Touché-2020	Quora	Avg.
ColBERT$_{v2}$/PLAID (k= 10)	33.3	69.0	15.3	34.5	25.6	85.1	43.8
ColBERT$_{v2}$/PLAID (k= 100)	33.4	69.2	15.3	35.4	25.2	85.4	44.0
ColBERT$_{v2}$/PLAID (k= 1000)	33.5	69.2	15.3	$\underline{35.5}$	25.6	$\underline{85.5}$	44.1
ColBERT$_{v2}$/WARP (n_{probe} = 32)	$\underline{34.6}$	$\underline{70.6}$	$\underline{16.2}$	33.6	$\underline{26.4}$	84.5	$\underline{44.3}$

Table 5: ColBERTv2/WARP nDCG@10 on BEIR. The last column shows the average over 6 BEIR datasets.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009