QBlock and GRABS: Memory-Efficient Learned Sparse Retrieval via Quantized Weight Blocks and Greedy Block Selection

Anonymous Author(s)

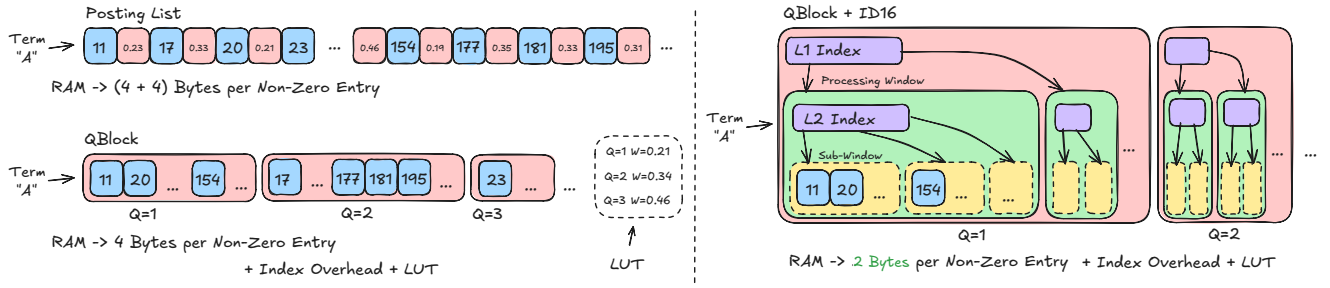

Figure 1: Memory Layout of Posting list, QBlock, QBlock + ID16

Abstract

Learned sparse retrieval (LSR) enables effective semantic search with inverted indices, but in performance-critical deployments the RAM budget available for memory- (or cache-) resident posting lists is often the limiting factor. We propose **QBlock**, a memory-efficient posting-list format that groups postings into *quantized weight blocks*, eliminating per-posting weight storage and exposing traversal units with homogeneous gains. At query time, we introduce **GRABS**, a **GR**eedy **A**daptive **B**lock **S**election algorithm that unifies query-term and posting-list pruning by selecting blocks under a fixed budget using a gain-based criterion. To further reduce memory footprint, we add a *sub-window* DocID re-encoding scheme that compresses 32-bit identifiers to 16-bit local IDs while supporting cache-efficient windowed accumulation. We evaluate QBlock +GRABS in a BigANN-style benchmark pipeline on MS-Marco (V1/V2) and NQ where latency, recall, RAM footprint and index construction time are evaluated. Across datasets, QBlock +GRABS achieves the best latency among *PartialDP* inverted-index baselines at fixed Recall@10 targets (0.91–0.99). While remaining a reasonable index construction time, QBlock substantially reduces the resident RAM footprint of the inverted index by eliminating per-posting weights (and optionally compressing DocIDs).

CCS Concepts

• **Information systems** → **Retrieval models and ranking**.

Keywords

Learned sparse representations, maximum inner product search, inverted index.

ACM Reference Format:

Anonymous Author(s). 2026. QBlock and GRABS: Memory-Efficient Learned Sparse Retrieval via Quantized Weight Blocks and Greedy Block Selection. In *Proceedings of Proceedings of the 49th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '26).* ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Semantic retrieval has become a core capability in modern web search. The shift is driven by retrieval-augmented generation (RAG), where retrieval quality directly bounds generation quality, and by broader adoption of semantic ranking in traditional applications such as e-commerce search and content feeds. In production, semantic retrieval typically encodes queries and documents into vector representations and retrieves candidates by similarity, often formulated as Maximum Inner Product Search (MIPS).

While dense embeddings are widely deployed, learned sparse retrieval (LSR) offers a complementary operating point. LSR models (e.g., DeepCT[9], SPLADE[12]) produce very high-dimensional sparse vectors (often 30K+ dimensions) where each dimension corresponds to a lexical term and the weight reflects its importance. This lexical-aligned structure enables inverted-index serving and interpretability, and inherits the scalability advantages of classical posting-list traversal. However, production systems face a higher bar: workloads must sustain tight latency under high QPS, and indices must scale to tens of millions to billions of documents under continuous ingestion. In this regime, memory footprint becomes a first-order bottleneck: RAM footprint limits how much of the index can remain resident per node and directly matters the hardware investment for a given data scale. Even within the cached scenario, improper RAM layout will lead to frequent cache eviction and harm the service latency with cache swap overhead. Meanwhile, CPU resource consumption during index construction can impact stability when indexing and querying co-occur in the same node. Although

SIGIR '26, Melbourne, VIC, Australia

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

this could be mitigated in the modern infrastructure where indices are built in a separate service fleet, it still matters how frequent the search engine can update the newly ingested documents.

Unlike approaches that achieve peak efficiency via heavy offline preprocessing and large auxiliary structures, this paper targets *memory-resident* learned sparse retrieval under limited RAM while preserving competitive effectiveness. Our starting point is an entry-wise view of sparse inner-product evaluation: for a fixed query term t, each posting entry contributes an additive score increment $gain(i, t; \mathbf{q}) = q_t x_{i,t}$, and each visited entry incurs one scatter-add update. Under a compute budget, approximate retrieval can therefore be cast as deciding *which posting regions to traverse* so as to collect high utility score mass with minimal wasted updates.

We implement this view with a format–algorithm co-design. During ingestion, QBlock quantizes document-side term weights and groups postings with the same quantized value into contiguous *weight blocks* (Figure 1). Each block stores only DocIDs plus lightweight metadata, while a small lookup table reconstructs a representative weight per bin. This achieves two effects simultaneously: (i) it removes the per-posting weight field for direct RAM reduction, and (ii) it converts each block into a *gain-homogeneous* traversal unit, since all entries in block (t, v) share the same per-entry gain $g = q_t \widehat{x}(v)$ for a given query. This differs from classic DocID-range blocking used by safe-pruning methods (e.g., MaxScore/WAND-style variants) [2, 33], which still require per-posting weights (or additional statistics) to evaluate contributions.

Given gain-homogeneous blocks, query-time pruning becomes a single global budget allocation problem rather than two independent heuristics. GRABS implements this by greedily prioritizing candidate blocks by their query-dependent gains and selecting a prefix under a stopping rule (e.g., time budgeting or α-mass thresholding). As a result, GRABS unifies *query-term pruning* and *posting-list truncation* into one mechanism that allocates computation across *both* query terms and weight ranges within each list. The selected blocks are then unfolded and executed with windowed scatter-add accumulation, preserving the PartialDP serving pipeline while reducing per-entry computation (no weight multiplication) and avoiding traversal of low-gain regions.

We further improve memory efficiency via DocID compression that integrates naturally with windowed processing. Specifically, we partition the corpus into sub-windows and re-encode global 32-bit DocIDs using 16-bit local IDs, enabling compact storage while maintaining fast scatter-add updates within cache-effective processing windows [22]. Finally, we introduce a mass-aware quantization strategy tailored to QBlock: instead of uniform binning, we place bin boundaries to concentrate precision on values that are both impactful (high weight) and frequently traversed under block selection, while avoiding excessive fragmentation into tiny blocks.

We evaluate QBlock +GRABS in a BigANN-style benchmark pipeline on MSMarco V1/V2 and NQ. Across datasets, QBlock substantially reduces the resident RAM footprint of the inverted index, and GRABS provides favorable latency–effectiveness trade-offs at fixed Recall@10 targets compared to standard PartialDP pruning baselines. In addition, compared to preprocessing-intensive sparse MIPS systems that invest heavily in clustering or auxiliary structures, QBlock +GRABS retains practical index construction cost while still delivering competitive query latency. The code implementation of this work is open-sourced[1].

We summarize the main contributions of this work as follows:

- **QBlock: Quantized block-based posting lists.** We design a posting-list memory layout that groups postings by quantized term weight into blocks, reducing per-posting storage and exposing block-level traversal units for efficient approximate sparse retrieval.
- **GRABS: Greedy block selection for approximate traversal.** We reformulate inverted-index pruning as a greedy block-selection problem using block gains to control latency and optimize the throughput–effectiveness trade-off.
- **Window / sub-window DocID compression.** We introduce a windowed re-indexing scheme that compresses 32-bit DocIDs to 16-bit local identifiers with practical evaluation efficiency, aligned with windowed PartialDP-style accumulation.
- **Data-driven quantization for QBlock.** We propose a mass-based quantization method with a reconstruction lookup table, biasing precision toward values that matter most under the block-pruned serving regime.

The rest of paper is organized as follows: Section 2 reviews related work. After a brief introduction to the preliminary in Section 3, Section 4 introduces QBlock and windowed processing, including DocID compression. Section 5 presents GRABS and its selection criteria. Section 6 describes the adaptive quantization method and its estimation procedure. Section 7 details experimental settings and results, including throughput–effectiveness trade-offs under limited-RAM constraints.

2 Related Work

Learned sparse retrieval models. Learned sparse retrieval (LSR) represents queries and documents as high-dimensional sparse vectors aligned with a vocabulary, enabling interpretability and compatibility with inverted-index serving. Early work such as DeepCT learns contextualized term weights while preserving posting-list execution [9]. The SPLADE family further advances sparse lexical expansion with extensive iterations on objectives, distillation, sparsity control, and efficiency/effectiveness trade-offs [12–15, 18–21]. Recent studies explore inference-free sparse retrievers and sparsification objectives for efficient serving [16, 30], and impact-style term weighting provides another compatible path for improving first-stage ranking within an inverted-index framework [26].

Query evaluation and pruning for inverted indexes. Efficient serving of learned sparse vectors depends not only on the representation but also on the query evaluation strategy (e.g., DAAT vs. SAAT), where SAAT-style scatter-add accumulation can be competitive or preferable under practical constraints [24]. A long-standing line of work improves exact top-k retrieval via *safe* pruning using score bounds and early termination, including MaxScore [33], WAND [2], and block-max variants such as BMW [10, 11]. In contrast, recent work targets *approximate* pruning under explicit latency or memory budgets, which is related to *anytime* retrieval in impact-ordered

[1] https://anonymous.4open.science/r/QBlock-4176

indexing [23] and commonly operates at block or superblock granularity, such as BMP and Dynamic Superblock Pruning [7, 27]. Alternative sparse MIPS designs such as SOSIA investigate hashing-based indexing with different memory–latency trade-offs [34].

System-level designs for approximate sparse MIPS at scale. Beyond pruning primitives, several systems co-design index structures and query processing for large-scale approximate sparse MIPS. SEISMIC and SEISMICWave reorganize posting lists with block structures and summaries (and optionally graph expansion) to enable aggressive skipping and high throughput [4, 5]. Sinnamon emphasizes robustness under streaming and continuously updated corpora [3], while SINDI [22] combines offline document-level pruning with query-time pruning heuristics and leverage SIMD for arithmetic acceleration. At the benchmark level, the BigANN sparse track shows that graph-based ANN indexes such as HNSW can also be applied to learned sparse embeddings (e.g., PyANN/GrassRMA-style systems) and can be competitive, underscoring the importance of end-to-end engineering for throughput at scale [6, 25, 31].

Inverted-index compression and memory efficiency. Throughput under limited RAM depends critically on inverted-index organization and compression, including DocID encoding and auxiliary metadata [35]. Classic `[docid, weight]` postings are simple but memory intensive when weights are stored per entry, motivating compact posting representations and integer coding techniques [28, 29].

Position of our approach. Our approach is most closely related to block-based approximate learned-sparse retrieval, but differs in both the *blocking signal* and the *pruning formulation*. QBlock constructs blocks by *quantized term weights*, eliminating explicit per-posting weight storage and yielding gain-homogeneous traversal units. At query time, GRABS recasts pruning as a *resource-budgeted block selection* problem, unifying query-term pruning and posting-list truncation into a single gain-based procedure that allocates computation across terms and weight ranges within each list.

3 Preliminary

Denoting $\mathcal{D} = \{\mathbf{x}_i\}_{i=1}^{N}$ as a corpus of N documents, where each document is represented as a high-dimensional sparse vector $\mathbf{x}_i \in \mathbb{R}^d_{\geq 0}$ with $\|\mathbf{x}_i\|_0 \ll d$. For a given query $\mathbf{q}$, the target of MIPS is retrieving $\mathcal{K}$ documents with the highest inner product:

$$\text{MIPS}(\mathbf{q}, K) \;=\; \text{argsort}^{K}_{i\in\{1,\ldots,N\}} \langle \mathbf{q}, \mathbf{x}_i \rangle \tag{1}$$

As a common practice inherited from keyword matching text search, an inverted index $\{L_j\}^d$ is usually employed for sparse vector indexing where each posting list is organized as:

$$L_j \;=\; \left\{ (i, x_{i,j}) \;\middle|\; i \in \{1,\ldots,N\},\; x_{i,j} \neq 0 \right\}. \tag{2}$$

Organized by non-zero terms, the inner product evaluation between $\mathbf{q}$ and different $\mathbf{x}_i$ can be repurposed into a traversal over corresponding posting lists:

$$a_i \;\leftarrow\; \sum_{j\in \text{nz}(\mathbf{q})} \sum_{(i,x_{i,j})\in L_j} q_j x_{i,j}. \tag{3}$$

Reported by previous works, the traversal process can be pruned according to query terms or posting list items for acceleration. In such cases the dot products are not fully evaluated for all candidates; only the most promising ones are considered. This family of methods is often referred to as partial dot-product (PartialDP).

4 QBlock: Block-based Posting List Layout

A standard sparse inverted index stores each posting as a `[docid, weight]` pair. For a corpus $\mathcal{D} = \{\mathbf{x}_i\}_{i=1}^{N}$ with sparse vectors $\mathbf{x}_i \in \mathbb{R}^d_{\geq 0}$, let L_j denote the posting list for term (dimension) j:

$$L_j \;=\; \{(i, x_{i,j}) \mid x_{i,j} \neq 0\}. \tag{4}$$

If each posting stores a document identifier of S_{doc} bytes and a weight of S_w bytes, the memory footprint of a fully memory-resident index can be approximated as

$$\text{RAM}_{\text{std}} \;\approx\; \sum_{j=1}^{d} |L_j|\,(S_{\text{doc}} + S_w) \;=\; \sum_{i=1}^{N} \sum_{j\in \text{nz}(\mathbf{x}_i)} (S_{\text{doc}} + S_w). \tag{5}$$

This section introduces QBlock, a posting-list layout that reduces per-posting storage while remaining compatible with PartialDP-style accumulation. QBlock has two components: (i) *quantized weight blocks* that eliminate explicit per-posting weight storage, and (ii) *sub-window DocID encoding* that compresses 32-bit DocIDs to 16-bit local IDs and aligns naturally with windowed query processing (as also discussed in SINDI [22]).

4.1 Quantized Posting List Blocking

We quantize posting weights into a small set of bins and group postings with the same quantized weight into contiguous blocks. Let $q(\cdot)$ map a real-valued weight to a discrete bin index $v \in \{0, \ldots, N_b\}$, and let $\widehat{x}(v) = LUT(v)$ be the reconstructed representative weight for bin v (the LUT construction is described in Section 6). For each posting list L_j, we define a block decomposition

$$L_j^{\text{block}} \;=\; \{B_{j,v}\}_{v<N_b}, \qquad B_{j,v} \;=\; \{\, i \mid (i, x_{i,j}) \in L_j,\; q(x_{i,j}) = v \,\}. \tag{6}$$

All postings within the same block $B_{j,v}$ share the same reconstructed document-side weight $\widehat{x}(v)$, and therefore share the same per-term gain for a fixed query. As a result, QBlock no longer needs to store a floating-point weight for each posting. Relative to the baseline in Eq. (5), this removes S_w bytes per posting, reducing the dominant term in RAM usage by a factor of

$$\frac{S_w}{S_{\text{doc}} + S_w}. \tag{7}$$

4.2 DocID Re-encoding with Windows

We further reduce per-posting storage by encoding document identifiers using 16-bit local IDs. We partition the global DocID space into *sub-windows* of size $W_s = 2^{16} = 65{,}536$. A document with global identifier d belongs to sub-window $k = \lfloor d/W_s \rfloor$ with local identifier $\ell = d - kW_s \in [0, W_s)$. Decoding is therefore

$$d \;=\; k \cdot W_s + \ell. \tag{8}$$

With this encoding, each posting stores ℓ as a `uint16` rather than d as a `uint32`. Combining *(i)* the removal of per-posting weights and *(ii)* 16-bit DocIDs yields a clear byte-level comparison: a standard `[uint32 docid, float32 weight]` posting uses 8 bytes, while

QBlock stores only a `uint16` local ID (2 bytes) per posting, plus small shared metadata.

4.3 Processing Windows for Cache-Resident Accumulation

With scatter-add as the score accumulation scheme, windowed index processing is a common technique to bound the score buffer and improve cache locality. Following the general practice discussed in SINDI [22], we process the corpus in *processing windows* sized such that the window-local score buffer remains cache-resident.

A naive choice is to use a sub-window size of $W_s = 65{,}536$ (enabling uint16 local DocIDs). However, on modern server CPUs this window is typically *too small* for efficient query processing: it increases the number of outer-loop iterations and amplifies per-window setup overhead (e.g., block pointer initialization and bookkeeping). In practice, we find that the "sweet spot" occurs when the score buffer occupies roughly $1\times \sim 2\times$ the per-core L2 cache capacity, offering reasonable setup overhead while maintaining good locality for random scatter-add updates. For example, consider a CPU core with a 1 MB L2 cache and a float32 score buffer (4 bytes per document). A score buffer of 2 MB corresponds to approximately $2 \cdot 2^{20}/4 \approx 5 \times 10^5$ documents, suggesting an effective processing window size on the order of $W \approx 500$K documents.

To reconcile the need for (i) larger cache-effective processing windows and (ii) 16-bit local DocID encoding, we group multiple sub-windows into a single processing window(the right part of Figure 1). Specifically, we combine m consecutive sub-windows into one processing window of size

$$W \;=\; m \cdot W_s. \tag{9}$$

In our implementation, we target a processing window size $\hat{W}$ by setting $m = \arg\min_m |\hat{W} - mW_s|$, which empirically balances these effects on typical server CPUs.

Within each processing window, we accumulate partial dot products into a window-local score buffer and maintain global top-K results by merging window-local candidates using a heap across windows.

5 GRABS: Search Pruning as Greedy Block Selection

Many pruning methods accelerate inverted-index retrieval by reducing the volume of postings evaluated. Two widely adopted mechanisms are (i) *query-term pruning*, which limits the number of query terms traversed, and (ii) *posting-list pruning*, which truncates or selectively traverses each posting list. With QBlock, these two mechanisms can be expressed within a single decision problem: *selecting which QBlock blocks to traverse* under a resource constraint.

5.1 Problem Setup

Recall that QBlock partitions each posting list (term) into blocks indexed by quantized weight bins. Each block can be represented as a pair (t, v) where t is the term (posting list) and v is the quantized weight bin. We denote such a block as $B_{t,v}$ and its postings as local DocIDs in the corresponding windows (Section 4.3). Given a query $\mathbf{q}$, the candidate block set touched by the query is

$$C(\mathbf{q}) \;=\; \{\, B_{t,v} \mid t \in \mathrm{nz}(\mathbf{q}) \,\}. \tag{10}$$

Given a selected subset $\mathcal{R} \subseteq C(\mathbf{q})$, the approximate retrieval score is computed by traversing blocks in $\mathcal{R}$ and accumulating partial dot products via scatter-add, window by window.

5.2 Block Utility and Gain-Based Selection

We motivate greedy block selection from an entry-wise view of sparse inner-product evaluation. Each posting entry $(i, x_{i,t})$ incurs one scatter-add update and contributes an incremental score:

$$\mathrm{gain}(i, t; \mathbf{q}) \;=\; q_t \cdot x_{i,t}, \tag{11}$$

Under a fixed traversal budget (bounded number of posting updates), a natural objective is to maximize the accumulated score contribution of the visited entries, for which the optimal greedy rule is to visit entries in descending $\mathrm{gain}(i, t; \mathbf{q})$.

QBlock groups postings by quantized weights, so all entries in the same block $B_{t,v}$ share a reconstructed weight $\widehat{x}(v)$ and therefore the same per-entry gain:

$$gain(B_{t,v}, \mathbf{q}) \;\triangleq\; q_t \cdot \widehat{x}(v). \tag{12}$$

As a result, prioritizing blocks by descending $gain(B, \mathbf{q})$ is equivalent to the entry-wise "highest-gain-first" traversal, while enabling efficient block-level iteration and pruning.

Latency model. We employ windowed processing (Section 4.3) with processing window size W and window set $\mathcal{W}$, where $M = |\mathcal{W}| = \lceil N/W \rceil$. Traversing a block incurs (i) per-block fixed overhead including iterator setup and cache miss, in which cache miss overhead correlates with window settings (W), skipping the window which doesn't contain the block, (ii) a per-posting cost dominated by scatter-add updates. We model the total traversal cost of block B across all windows as

$$\mathrm{Latency}(B) \;=\; \sum_{w \in \mathcal{W} s.t. B \in w} c_{\mathrm{blk}}(W) \;+\; \gamma\,|B|. \tag{13}$$

Here $c_{\mathrm{blk}}(W)$ is a fixed per-block cost and γ is the per-posting cost. The coefficient γ can be estimated by profiling blocks of different sizes. Function $c_{\mathrm{blk}}(W)$ is observed to be non-linear. Instead of parameterizing it, it is suggested to profile with a fixed window size and treat it as a constant.

5.3 GRABS Algorithm

GRABS greedily selects blocks according to a gain-based criterion and stops when a selection condition is satisfied (Section 5.4). After selection, each selected block will be unfolded and scatter-add will be performed to collect the score along the document dimension. The algorithm is defined as Algorithm 1.

5.4 Selection Conditions

The function BlockSelect in Algorithm 1 greedily collects blocks starting from the largest gain mass $v(B)$ until a stop condition is met. We consider two stop conditions targeting different operational objectives: (i) *time budgeting* to provide stable latency, and (ii) *α-mass thresholding* which adapts to query difficulty while controlling global resource investment.

5.4.1 Time Budgeting. Given a latency budget T_{bgt}, we estimate the end-to-end query processing cost for a selected block set $\mathcal{R}$ as

$$\text{Latency}(\mathbf{q}) \;=\; c_{qry} \;+\; \sum_{B\in\mathcal{R}} \text{Latency}(B) \;+\; c_{rnk}, \tag{14}$$

where c_{qry} contains a per-query setup overheads such as query parsing and block selection, and c_{rnk} is the re-ranking cost, both can be simply estimated by profiling. Please note that c_{rnk} may vary across different queries, but the budget can still be achieved by employ P90 or P99 of c_{rnk} as the bound. During greedy selection, blocks are added in decreasing gain-mass order and the selection terminates once the estimated latency exceeds T_{bgt}. In contrast to methods that stop traversal once the time budget is depleted [3], we estimate latency *before* execution to keep PartialDP accumulation symmetric across windows and ensure score comparability.

5.4.2 α-mass Thresholding. Empirically, gain mass often exhibits heavy-tailed concentration: a small fraction of blocks accounts for most of the cumulative mass. This motivates an adaptive selection rule that traverses fewer blocks for "easy" queries while allocating more compute to "hard" queries. Let C be the candidate blocks for $\mathbf{q}$ and define the total gain-mass as $S = \sum_{B\in C} gain(B,\mathbf{q})$. We select the smallest prefix (in descending $gain(B,\mathbf{q})$ order) whose cumulative mass reaches an α fraction of S:

$$\mathcal{R} \;=\; \arg\min_{\mathcal{R}'\subseteq C} \left\{ |\mathcal{R}'| \;:\; \sum_{B\in\mathcal{R}'} v(B) \;\geq\; \alpha \sum_{B\in C} v(B) \right\}, \tag{15}$$

where $\mathcal{R}'$ is restricted to be a prefix of the gain-mass-sorted list. This rule adapts traversal depth to each query while maintaining a global control parameter α.

6 Adaptive Quantization for Blocking

A simple baseline for weight quantization is uniform (equal-width) binning. While straightforward, uniform binning is not optimized for block-based traversal: it may create many small blocks (high fixed overhead) and wastes precision on values that are rarely traversed after block selection. We therefore design a quantizer specifically for quantized-block posting lists and greedy block selection. A practical quantizer should:

- **Avoid producing many small blocks.** Each block introduces fixed traversal overhead, so overly fine bins can be inefficient.
- **Allocate more precision to high-impact values.** Larger weights contribute more to dot products and should be represented with finer granularity.
- **Allocate more precision to frequently traversed values.** Under block selection, many low-weight postings are rarely visited; allocating bins to them yields little benefit.

6.1 Mass-aware Binning

Let $v \in \{0,\dots,255\}$ denote the pre-quantized 8-bit weight value (e.g., after a global rescaling of raw term weights). Let $h(v)$ be the number of postings in the corpus whose pre-quantized value equals v. To capture both contribution and traversal likelihood, we assign each value v a *mass* score:

$$mass(v) \;=\; v \cdot h(v) \cdot p(v), \tag{16}$$

where $p(v) \in [0,1]$ estimates the probability that postings with value v are traversed (i.e., not pruned) under the block selection policy. Given a target number of bins B (not necessarily a power of two), we construct a quantizer $q(\cdot)$ by *balanced mass binning*: we partition the ordered domain $\{0,\dots,255\}$ into B contiguous bins such that each bin contains approximately equal total mass. This concentrates quantization resolution on values that are both important (large v) and frequently traversed (large $p(v)$), while avoiding excessive fragmentation into small blocks.

Algorithm 1 GRABS retrieval on QBlock with processing windows

Require: Query $\mathbf{q}$; block collection $\mathcal{B}$; processing windows $\mathcal{W}$ of size W; top-K
Ensure: Top-K documents w.r.t. approximate scores

$C \leftarrow \{ B \in \mathcal{B} \mid term(B) \in \text{nz}(\mathbf{q}) \}$ ▷ candidate blocks touched by query terms
for all $B \in C$ **do**
  $gain(B,\mathbf{q}) \leftarrow q_{term(B)} \cdot \widehat{x}(bin(B))$ ▷ Eq. (12)
end for
$C \leftarrow \text{SortDesc}(C,\ gain)$
$\mathcal{R} \leftarrow \text{BlockSelect}(C,\ gain)$ ▷ stop by budget or α-mass
$H \leftarrow$ min-heap of capacity K
for all $\omega \in \mathcal{W}$ **do** ▷ ω is a window instance; W is window size
  initialize $score[0{:}W) \leftarrow 0$
  for all $B \in \mathcal{R}$ **do**
    $g \leftarrow g(B,\mathbf{q})$
    for all $\ell \in postings(B,\omega)$ **do** ▷ local IDs of B that fall in window ω
      $score[\ell] \leftarrow score[\ell] + g$
    end for
  end for
  for $\ell = 0$ to $W-1$ **do**
    if $score[\ell] > 0$ **then**
      $doc \leftarrow offset(\omega) + \ell$ ▷ local ID → global ID
      $\text{HeapPush}(H,\ (doc, score[\ell]))$
    end if
  end for
end for
return re-rank(H) ▷ re-ranking by dot production

Remark. The routine $postings(B,\omega)$ can be implemented via a compact second-level pointer structure that stores, for each block B, the begin/end offsets of its postings within each processing window ω, avoiding explicit set intersections at query time.

6.2 Reconstruction via Lookup Table

Given the adaptive quantizer $q(v) = \tilde{v}$ with $\tilde{v} \in \{0,\dots,B-1\}$, we store a representative reconstructed weight for each bin in a lookup table:

$$LUT(\tilde{v}) \;=\; \frac{\sum_{v:\, q(v)=\tilde{v}} h(v)\cdot v}{\sum_{v:\, q(v)=\tilde{v}} h(v)}. \tag{17}$$

At query time, each block $B_{t,\tilde{v}}$ uses $\widehat{x}(\tilde{v}) = LUT(\tilde{v})$ as its shared document-side weight. The memory overhead of the LUT is $O(B)$, negligible relative to posting storage.

6.3 Selectivity Estimation

The term $p(v)$ in Eq. (16) models how likely a value v is to be traversed under the block selection policy. Directly computing $p(v)$ is difficult because selection depends on the competitive landscape among blocks across all query terms. We therefore use a lightweight approximation based on a random cutoff model.

Let β be a random threshold such that postings with $v \geq \beta$ are likely to be traversed, while those with $v < \beta$ are typically skipped. Define the survival event $s(v;\beta) = \mathbb{I}(v \geq \beta)$. Assuming $\beta \sim \mathcal{N}(\mu, s^2)$, we obtain

$$p(v) \;=\; \mathbb{E}_\beta[s(v;\beta)] \;=\; \Pr(v \geq \beta) \;=\; \Phi\Big(\frac{v-\mu}{s}\Big), \qquad (18)$$

where $\Phi(\cdot)$ is the standard normal CDF. In practice, (μ, s) can be estimated from sampled queries by observing the effective cutoff values implied by selected blocks. Empirically, because Eq. (16) already includes the factor v, we bias (μ, s) toward smaller values to further suppress bins that are almost never traversed. The setting of $p(v)$ will also incur a built-in prune by leaving out those entries with $\tilde{v} = 0$.

7 Experiments

7.1 Setup

Datasets and Models. We evaluate GRABS with QBlock on three publicly available retrieval datasets. As our primary testbed, we use the MSMarco passage ranking V1 corpus [1], which contains 8.8M passages and uses the 6,980 MS MARCO dev queries for evaluation. To assess scalability, we additionally use the MSMarco passage ranking V2 corpus [8], which expands the collection to 137M passages. Finally, we include Natural Questions (NQ)[17] in BEIR [32] as an evaluation in the question answering retrieval scenario.

Unless otherwise stated, all documents and queries are encoded with the *CoCondenser-EnsembleDistil* learned sparse model [14], producing non-negative sparse vectors over a 30,522-term vocabulary. Table 2 summarizes the sparsity statistics of the resulting representations, which directly determine posting list lengths and block size distributions. In addition, we report results with a query-time inference-free sparse encoder, NeuralSparse-v3-GTE [16] [2], where query vectors are produced via tokenization and a lightweight weight lookup without running a neural query encoder.

Baselines. We compare GRABS + QBlock with three groups of baselines that cover standard inverted-index PartialDP traversal, systems-optimized PartialDP variants, and non-PartialDP ANN indexes:

- **PartialDP on inverted indexes with pruning.** These baselines use conventional inverted-index scoring (PartialDP) and accelerate query processing using two widely adopted pruning strategies: *(i) query-term pruning*, which retains only the top-T query terms by query weight at the query-time. *(ii) posting-list pruning*, which traverses only a prefix (top ranked value entries) of each posting list. (ii) is a indexing-time pruning method which will affect index size. In our experiment, we adopt α-mass to truncate. We sweep the pruning parameters (T and α) to obtain recall–latency trade-off curves.
- **Systems-optimized PartialDP.** We include LinScan [3] where we direct sweep the time-budget for the trade curve, and SINDI [22], which combines indexing-time pruning and query-time pruning together. Here, we employ several document pruning levels and the performance is tuned by the query pruning parameter.
- **Non-PartialDP ANN indexes.** We additionally evaluate approaches that depart from standard PartialDP traversal. Two state-of-the-art sparse retrieval systems based on block skipping are included: SEISMIC [4] and Super-Blocks [7]. For SEISMIC, we will control the RAM footprint by adjusting λ – number of document participating the clustering in each posting list, while we sweep the *query_cut* parameter at runtime for different recall levels. In Super-Blocks, the index RAM footprint is not tunable and retrieval related parameters (β, μ, and η) are tuned for different target recalls [7].

For PartialDP baselines, we implement the methods in C++ following the algorithmic descriptions in the corresponding papers, to have a fair comparison, we also developed its variant with windowed index processing. For methods with public implementations, we use the authors' open-source code for evaluation.[3] The weight precision for all the evaluated methods are set to `FLOAT32`, leading to a unified forward-index size of 5.96 GB for MSMarco V1, 99.43 GB for MSMarco V2 and 2.22 GB for NQ.

QBlock + GRABS Settings & Variants. In all the experiments except quantization ablation, we adopt a quantizer with 16 bins which is consider to be the most balanced choice between efficiency and relevance. We evaluate two GRABS block selection policies: *time budgeting* and *α-mass thresholding* (Section 5.4). We further ablate key design if DocID re-encoding (ID16) which directly impact the efficiency. Finally, an optional integration with document-level pruning is also evaluated to test whether additional skipping can further reduce traversal cost without harming effectiveness. For MSMarco V1 and NQ dataset, we pick the best performing $k' = 500$ as default, while in MSMarco V2 k' is scaled up to 3000 (check Section 7.3 for details).

Evaluation Metrics. Our first comparison metric is *efficiency at fixed effectiveness*. Specifically, we report the average query latency at several fixed Recall@10 targets (0.91–0.99) by sweeping pruning/selection parameters. For memory, we measure the *resident RAM footprint on serving nodes* for the retrieval index, including posting storage and auxiliary metadata (e.g., block sizes for QBlock, and cluster summary vectors for SEISMIC). This metric reflects common production practice where indexes are constructed offline (or on dedicated builder nodes) and then dispatched to serving fleets; therefore, we focus on the final resident footprint rather than transient peak allocations during construction. Also, we *do not* count in the forward-index size and focus solely on the inverted index, since the goal is to compare indexing and search strategies with an unified vector storage. The index construction time is also evaluated

[2] `https://huggingface.co/opensearch-project/opensearch-neural-sparse-encoding-doc-v3-gte`

[3] LinScan: `https://github.com/pinecone-io/research-bigann-linscan`;
SINDI: `https://github.com/antgroup/vsag`;
SEISMIC: `https://github.com/TusKANNy/seismic`;
Super-Blocks: `https://github.com/thefxperson/hierarchical_pruning`.

	MSMarco V1							MSMarco V2							NQ						
	inv. idx. RAM	Const. Time	Latency (μs) when Recall@10 =					inv. idx. RAM	Const. Time	Latency (μs) when Recall@10 =					inv. idx. RAM	Const. Time	Latency (μs) when Recall@10 =				
Approaches	(GB)	(s)	0.91	0.93	0.95	0.97	0.99	(GB)	(s)	0.91	0.93	0.95	0.97	0.99	(GB)	(s)	0.91	0.93	0.95	0.97	0.99
PartialDP																					
↪ query prune	8.09	19.5	13711	14537	16145	17934	21418	135.1	298.7	207135	221534	241092	264728	355210	3.0	6.7	5817	6240	6637	7101	8112
↪ pst. list prune	0.895†	81.5	24175	25431	27702	31510	37724	15.1†	673.2	202411	219689	243521	286067	381342	0.4†	35.7	9328	9932	10208	11228	12183
↪ query prune (win.)	8.15	40.0	3909	4302	5131	6396	8603	136.0	640.9	54329	61142	72593	85871	146822	3.1	15.1	2907	3232	3489	3973	4901
↪ pst. list prune (win.)	0.953†	83.0	10230	11527	13109	16092	22101	16.0†	703.9	67421	75534	86175	107911	171349	0.4†	36.5	7067	7671	8589	9907	12503
LinScan	8.09	19.5	16717	17324	18303	20372	24401	135.1	298.7	239341	251497	270628	305331	387532	3.0	6.7	7892	8619	9170	10332	11704
SINDI																					
↪ w/o doc prune	7.91	40.0	2247	2610	3603	5056	5146	131.97	714.5	28278	37938	38249	63538	97847	2.94	14.3	1309	1395	1705	2727	2819
↪ doc prune ($\alpha = 0.5$)	1.51	49.0	1039	1048	1096	1228	1826	24.79	817.3	8085	8990	10282	13663	-	0.57	19.1	526	554	665	1148	1976
SEISMIC																					
↪ Plain	3.84	2672.0	233	268	358	567	-	56.87	26259.0	1169	1480	1870	3713	-	3.26	994.0	259	323	420	977	-
↪ int8 Summary	2.34	2791.0	247	289	384	596	-	33.28	25336.0	1164	1472	1891	3664	-	1.91	1015.0	272	337	441	1011	-
Super-Blocks	40.86	168	1750	2078	2489	2875	3038	755.6	3062.6	30775	34662	38317	41549	44840	16.10	63	1272	1476	1708	1988	2175
QBlock + GRABS																					
↪ Time bgt.	1.68	41.2	606	695	838	1131	1802	27.16	897	7002	7974	10008	13062	19542	0.68	16.49	394	449	533	688	1223
↪ α-mass			581	662	767	957	1568			6941	7753	9001	11355	18997			389	444	536	683	1096
↪ ID16	1.05	54.5	647	747	858	1090	1777	16.68	1075	7712	8801	9792	12781	21783	0.41	21.34	493	582	742	981	1657
↪ doc prune ($\alpha = 0.7$)	1.37	70.3	592	675	782	975	1515	22.43	1433	7036	7907	8923	11404	19011	0.52	27.53	388	447	535	679	1027
↪ doc prune ($\alpha = 0.5$)	0.84	55.7	574	645	733	917	1670	13.51	1112	6778	7301	8643	10709	18520	0.32	21.85	383	436	520	666	1314
↪ doc prune ($\alpha = 0.5$) + ID16	0.61	70.2	635	716	835	1050	1921	9.54	1397	7593	8369	9621	12292	22154	0.23	24.88	440	509	681	901	1938

†: RAM foot print is calculated with pruning parameter Recall@10=0.95

Table 1: Comparison on Key Metrics between QBlock + GRABS and baselines with unlimited resource budget.

Embedding	Dataset	#nz per doc	#docs	#nz per query	#queries
CoCondenser-EnsembleDistil	MSMarco V1	119.96	8.8M	43.95	6980
	MSMarco V2	127.94	138M	44.81	3903
	NQ	174.52	3.7M	46.97	3500
NeuralSparse-v3-GTE	MS Marco V1	180.41	8,8M	6.89	6980

Table 2: Sparse embedding statistics of CoCondenser-EnsembleDistil on different datasets

to reflect the efficiency upon document ingestion. In this paper, we meter the time period since all the vector data are loaded into RAM from disk until the index is prepared ready for query.

Hardware & Execution Settings. All experiments are conducted on a cloud compute instance with 64 vCPUs of 2.6GHz and 1.47 TB RAM, while all the evaluations are conducted with *single thread*, including both index construction and querying. The CPU is with a 2MB L2 cache per core thus the recommended processing window size is 1M documents.

7.2 Main Results

Table 1 summarizes the comparison between the proposed method with the baselines across effectiveness–efficiency, memory footprint, and index construction time. We highlight the key observations below and defer detailed ablations to Section 7.3.

Efficiency at fixed effectiveness. From Table 1, method of SEISMIC achieve the lowest latency at fixed Recall@10, which is expected because it invests substantial computation resource and RAM into building finer-grained index structures. This will be further elaborated in the next paragraph. Within the PartialDP family, GRABS + QBlock achieves the best latency at all the recall targets. When Recall@10=0.95 our approach (α-mass) reduces the latency by 95% over query-term pruning and by 78.9% over SINDI (plain version). These gains support our central claim that GRABS provides a unified and more adaptive mechanism than rigid query-term truncation or per-list truncation alone, by jointly allocating traversal budget across terms and gain-homogeneous regions exposed by QBlock. Switching to a scaled dataset (MSMarco V2), GRABS is still leading the PartialDP family on the efficiency.

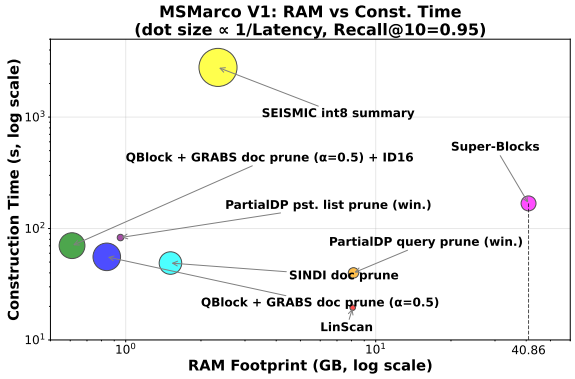

Figure 2: Scatter diagram comparing three metrics: RAM footprint, Index construction Time and Latency.

We also find that document-level pruning can not only reduce memory consumption but also improve the search efficiency. For example, with $\alpha = 0.5$, document pruning even improves QPS by 4.5% at Recall@10 = 0.95. This suggests that document-level pruning is complementary to GRABS: it reduces wasted scatter-add updates on low-utility documents, while controlling which gain-homogeneous regions of QBlock posting lists are traversed.

RAM footprint and Index Construction Time. Figure 2 presents a scatter diagram which positions RAM footprint, index construction time and search latency into a unified view, where the dot size reflects the search latency (bigger dot, lower latency). As the *x-axis* and *y-axis* represent the two most important factors in real production deployment, the dots lying at the bottom-left would be preferred. The two evaluated Non-PartialDP methods are locating at two extreme corners on the diagram. The SEISMIC algorithm will occupy 283% more RAM on inverted index than the most compact version of QBlock and requires 39.8× on the indexing time, allowing it to offer a 2.89× speed-up comparing to GRABS. Although with

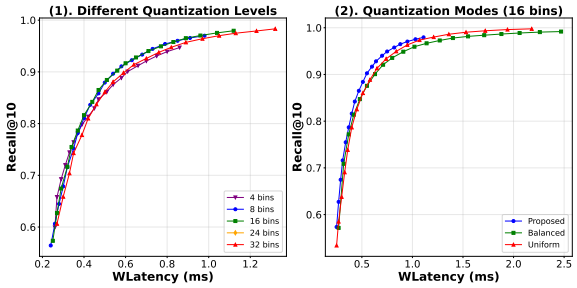

Figure 3: (1) Recall@10 vs. Latency comparison of different quantization levels (B). (2) Recall@10 vs. Latency comparison of different quantization methods.

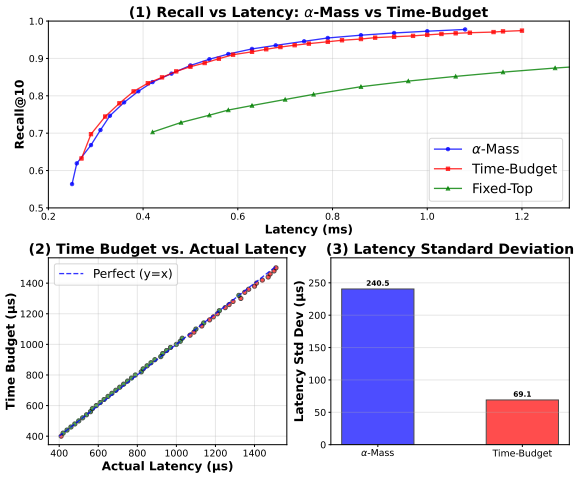

Figure 4: (1) Latency-Recall Comparison between α-mass and time-budget. (2) Relationship between time budget and latency. (3) Latency variance comparison.

comparable index construction time, Super-Blocks is consuming way more RAM than other methods (40.86GB).

Presented by dot locations, the QBlock + GRABS approaches are leading the low-cost quadrant — reasonable construction time, lowest RAM consumption and outperforming latency. It can also be observed that document ID re-encoding (ID16) approach can reasonably reduce the inverted index foot print (-37.5% compared with plain QBlock, see Table 1) [4], with limited trade-off on the latency. The hero-combination is ID16 + document pruning (green dot), which achieves a minimum RAM cost (for inverted index) of 0.61GB, and achieve a latency (Recal@10 = 0.95) of 835μs.

7.3 Ablations

α-mass vs. Time Budgeting. Figure 4.(1) compares the latency–recall curves of α-mass and time budgeting with more data points on MSMarco V1, which also involves a baseline method of constantly selecting N top-rated blocks. It can be observed that the proposed selection conditions outperform the non-adaptive selection method with margin. α-mass succeeds Time-budget a little bit in the effective recall zone (Recall@10 > 0.9). The advantage of time budgeting is providing a stable search experience; accordingly, query latency is more consistent (see Figure 4.(3)). The comparison between offered time budget and actual observed latency (average) is presented in Figure 4.(2). The tight dots distributed around the ideal correlation line indicates that the provided latency estimation model in Eq. 14 is accurate.

Approaches	Latency (μs) when Recall@10 = 0.91	0.93	0.95	0.97	0.99
PartialDP					
↪ query pruning (win.)	3102	3489	4064	5003	7237
↪ posting list pruning (win.)	2385	2704	2885	3401	4598
SINDI					
↪ w/o doc prune	1970	1982	2440	3089	5004
↪ doc prune (α = 0.5)	831	953	1044	1246	1773
QBlock + GRABS					
↪ Time bgt.	421	461	541	686	1008
↪ α-mass.	450	496	562	697	1021

Table 3: Comparison between our methods with representative baselines on inference-free embeddings. Dataset: MS-Marco V1, Model: NeuralSparse-v3-GTE.

Quantizations. Figure 3.(1) shows the throughput performance for different bin counts. We observe a clear effectiveness–efficiency trade-off: more quantization levels increase weight precision but introduce more loop overhead due to smaller blocks; fewer quantization levels yield larger blocks with lower overhead but reduce weight precision. We also observe from Figure 3.(2) that proposed adaptive quantization method outperforms two representative baselines: uniform quantization and balanced quantization (each bin contains the same $h(v)$). This proves that the proposed design principles are reasonable.

Re-ranking. The size of candidate set for for re-ranking – k' – also has trade-off, that is, a small k' will have less re-ranking overhead but may repy on broaden block traverse to ensure the recall while a bigger k' will require less block traverse but cost more time on re-ranking. The curves in Figure 5 show the recall–latency trade-off with different values of k' on MSMarco V1 dataset and MSMarco V2 dataset. The curve distribution shows that there is one safe range for k' within which the performance is stable and either too small or too big values will lag on the effectiveness–efficiency. We can observe the best k' for MSMarco V1 is 500 and the best k' for MSMarco V2 is 3000. This derives that the best k' is positively related to the data size.

Inference-free. We also evaluate QBlock + GRABS with inference-free embeddings [16] — another important branch for LSR which simplifies the query embedding into tokenization for retrieval efficiency. Such embedding bring a different distribution where query vectors are extremely sparse (see Table 2). Table 3 compares the latency upon key Recall@10 points with representative baselines on MSMarco V1 data with NeuralSparse-v3-GTE embeddings. We observe that GRABS maintains the advantages over other PartialDP methods. For example, when Recall@10=0.95, the average latency of QBlock is 48.3% less than SINDI. It indicates that the GRABS design remains effective under inference-free sparse embeddings.

[4]The reduction is not ideal 50% because the window index overhead is around 100MB, and another 160MB is from ID16 extra sub-window indexing. Values from MSMarco V1 dataset.

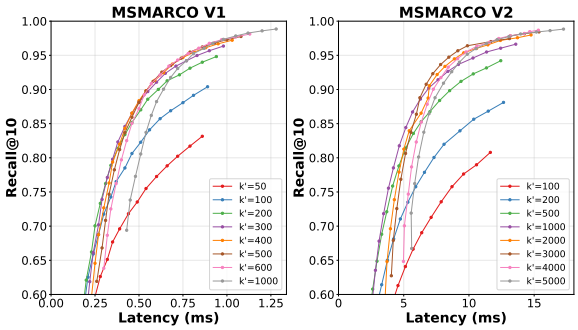

Figure 5: Recall@10 vs. Latency comparison over different k' values

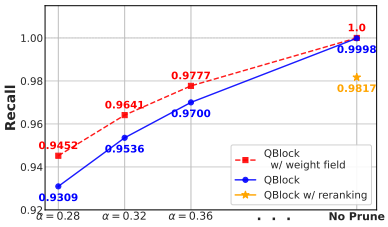

Figure 6: Recall@10 Comparison between QBlock and its variant version with weight field.

7.3.1 Investigations. In this section we conduct deeper investigations on ablations regarding the following research questions:

(RQ1) What is the impact of quantization error? We conduct an ideal experiment to measure the impact of quantization loss: we retain the original weight field in the index and perform unquantized scatter-add after QBlock block selection. In this case, the accuracy is only related to block selection (pruning). Figure 6 compares Recall@10 between QBlock and its variants under various α settings. The orange star shows the recall lost of 1.88% purely brought by quantization. It can be fully mitigated by re-ranking. The margin between QBlock and QBlock + weight field grows when smaller α is applied. This shows that quantization error and pruning loss are not impacting individually, instead, aggressive pruning is "amplifying" the quantization error.

(RQ2) Is the trained quantizer generalizable? Since the proposed quanization methods is data-driven, it is naturally concerned if the trained quantizer generalizable to new data. Figure 7.(1) shows the comparison on NQ dataset with quantizers trained from MSMarco V1 dataset and NQ dataset itself. The NQ quantizer outperforms MSMarco V1 quantizer negligibly, suggesting the quantization method is generalizable across datasets.

(RQ3) Is blocking itself harming the cache locality? A natural suspicion is that the block-wise overhead c_{blk} reflects not only loop overhead but also potential loss of cache locality. In particular, SINDI traverses posting lists in increasing DocID order, which may have a more regular access pattern to the score buffer, while QBlock may jump across DocIDs when iterating over blocks. We conduct an ideal experiment that shuffles document IDs within all blocks. Figure 7.(2) compares the recall–latency curve of the original indices

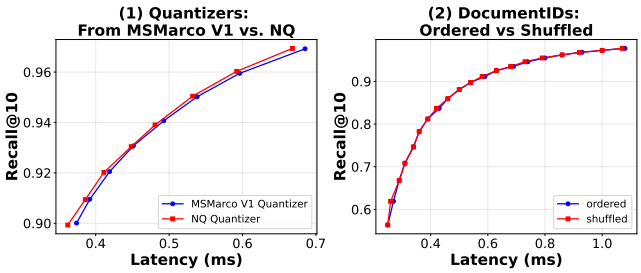

Figure 7: (1). The Recall@10 vs. Latency Comparison between quantizer trained from MSMarco V1 and quantizer trained from NQ. Conducted on NQ dataset. (2). Comparison between ordered document ID in block vs. shuffled document ID in block.

and shuffled ones. We find that shuffling has no measurable impact on performance. Also, we measure the CPU cache miss rate using profiling tool and the result is 2.98% vs. 3.20%. These evidences suggest the DocID ordering within traversed postings is not the bottleneck in our setting.

8 Conclusion

We studied memory-resident learned sparse retrieval under strict RAM and latency constraints, and proposed a unified design that reduces index footprint while improving query-time efficiency–effectiveness trade-offs. Our approach combines two components: QBlock, a quantized block-based posting-list format that removes per-posting weight storage by grouping postings into gain-homogeneous blocks; and GRABS, a greedy block selection algorithm that unifies query-term and posting-list pruning as a single resource-budgeted decision.

The key insight is that blocking by *quantized weight* exposes a natural traversal unit for approximate sparse MIPS. Because all postings in a QBlock block share a reconstructed weight, each block has constant per-posting gain for a query, enabling direct prioritization of traversal under a budget. This aligns with windowed scatter-add accumulation; combined with sub-window DocID re-encoding, it further reduces per-posting storage while preserving cache-effective processing.

Across MSMarco V1/V2 and NQ, QBlock + GRABS achieves the state-of-the-art latency among PartialDP method family at Recall@10 targets 0.91 ∼ 0.99 and lead the low cost quadrant when comparing RAM footprint and index construction time. Ablations indicate that gain-mass selection is an effective default, mass-aware quantization concentrates the best precision where it matters under pruning, and the approach remains effective with inference-free sparse queries.

Two future directions appear especially promising. First, block selection can be optimized beyond greedy gain mass via learnable criterion. Second, extending QBlock + GRABS to dynamic and streaming corpora—supporting incremental updates, stable block maintenance, and periodic or online re-quantization.

References

[1] Payal Bajaj, Daniel Campos, Nick Craswell, Li Deng, Jianfeng Gao, Xiaodong Liu, Rangan Majumder, Andrew McNamara, Bhaskar Mitra, Tri Nguyen, Mir

Rosenberg, Xia Song, Alina Stoica, Saurabh Tiwary, and Tong Wang. 2016. MS MARCO: A Human Generated MAchine Reading COmprehension Dataset. *arXiv preprint arXiv:1611.09268* (2016). doi:10.48550/arXiv.1611.09268

[2] Andrei Z. Broder, Shlomo Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. 2003. Efficient Query Evaluation using a Two-Level Retrieval Process. In *Proceedings of the 12th International Conference on Information and Knowledge Management (CIKM 2003)*. ACM, 426–434. doi:10.1145/956863.956944

[3] Sebastian Bruch, Franco Maria Nardini, Amir Ingber, and Edo Liberty. 2023. An Approximate Algorithm for Maximum Inner Product Search over Streaming Sparse Vectors. *ACM Transactions on Information Systems* (2023). arXiv:2301.10622 [cs.IR] doi:10.1145/3609797

[4] Sebastian Bruch, Franco Maria Nardini, Cosimo Rulli, and Rossano Venturini. 2024. 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 (SIGIR '24)*. ACM, 152–162. doi:10.1145/3626772.3657769

[5] Sebastian Bruch, Franco Maria Nardini, Cosimo Rulli, and Rossano Venturini. 2024. Pairing Clustered Inverted Indexes with κ-NN Graphs for Fast Approximate Retrieval over Learned Sparse Representations. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management (CIKM)*. arXiv:2408.04443 [cs.IR] doi:10.1145/3627673.3679977

[6] Sebastian Bruch, Franco Maria Nardini, Cosimo Rulli, Rossano Venturini, and Leonardo Venuta. 2025. Investigating the Scalability of Approximate Sparse Retrieval Algorithms to Massive Datasets. In *Proceedings of the European Conference on Information Retrieval (ECIR)*. arXiv:2501.11628 [cs.IR]

[7] Parker Carlson, Wentai Xie, Shanxiu He, and Tao Yang. 2025. Dynamic Superblock Pruning for Fast Learned Sparse Retrieval. *CoRR* abs/2504.17045 (2025). arXiv:2504.17045 doi:10.48550/arXiv.2504.17045

[8] Nick Craswell, Bhaskar Mitra, Emine Yilmaz, Daniel Campos, and Jimmy Lin. 2021. Overview of the TREC 2021 Deep Learning Track. In *Proceedings of the Thirtieth Text REtrieval Conference (TREC 2021)*. https://trec.nist.gov/pubs/trec30/papers/Overview-DL.pdf

[9] Zheng Dai and Jamie Callan. 2020. Context-Aware Term Weighting For First Stage Passage Retrieval. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2020, Xi'an, China, July 25-30, 2020*, Jimmy Lin, Xueqi Cheng, Joemon M. Jose, Jian-Yun Nie, Leonhard Hennig, Mauro Castelli, Hideo Joho, Brian Davis, Guglielmo Faggioli, and Nicola Ferro (Eds.). ACM, 1533–1536. doi:10.1145/3397271.3401290

[10] Constantinos Dimopoulos, Sergey Nepomnyachiy, and Torsten Suel. 2013. Optimizing top-k document retrieval strategies for block-max indexes. In *Sixth ACM International Conference on Web Search and Data Mining (WSDM)*. 113–122. doi:10.1145/2433396.2433412

[11] Shuai Ding and Torsten Suel. 2011. Faster top-k document retrieval using block-max indexes. In *Proceeding of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2011, Beijing, China, July 25-29, 2011*, Wei-Ying Ma, Jian-Yun Nie, Ricardo Baeza-Yates, Tat-Seng Chua, and W. Bruce Croft (Eds.). ACM, 993–1002. doi:10.1145/2009916.2010048

[12] Thibault Formal, Carlos Lassance, Benjamin Piwowarski, and Stéphane Clinchant. 2021. SPLADE: Sparse Lexical and Expansion Model for First Stage Ranking. *arXiv preprint arXiv:2107.05720* (2021). https://arxiv.org/abs/2107.05720

[13] Thibault Formal, Carlos Lassance, Benjamin Piwowarski, and Stéphane Clinchant. 2021. SPLADE v2: Sparse Lexical and Expansion Model for Information Retrieval. *arXiv preprint arXiv:2109.10086* (2021). https://arxiv.org/abs/2109.10086

[14] Thibault Formal, Carlos Lassance, Benjamin Piwowarski, and Stéphane Clinchant. 2022. 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 (SIGIR '22)*. ACM, 2353–2359. doi:10.1145/3477495.3531857

[15] Thibault Formal, Carlos Lassance, Benjamin Piwowarski, and Stéphane Clinchant. 2024. Towards Effective and Efficient Sparse Neural Information Retrieval. *ACM Transactions on Information Systems* (2024). doi:10.1145/3634912

[16] Zhichao Geng, Yiwen Wang, Dongyu Ru, and Yang Yang. 2025. Towards Competitive Search Relevance for Inference-Free Learned Sparse Retrievers. *arXiv preprint arXiv:2411.04403* (2025). https://arxiv.org/abs/2411.04403

[17] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, Kristina Toutanova, Llion Jones, Matthew Kelcey, Ming-Wei Chang, Andrew M. Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. 2019. Natural Questions: A Benchmark for Question Answering Research. *Transactions of the Association for Computational Linguistics* 7 (2019), 452–466. doi:10.1162/tacl_a_00276

[18] Carlos Lassance and Stéphane Clinchant. 2022. An Efficiency Study for SPLADE Models. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '22)*. ACM, 2220–2226. doi:10.1145/3477495.3531833

[19] Carlos Lassance, Hervé Déjean, Stéphane Clinchant, and Nicola Tonellotto. 2024. Two-Step SPLADE: Simple, Efficient and Effective Approximation of SPLADE. *arXiv preprint arXiv:2404.13357* (2024). https://arxiv.org/abs/2404.13357

[20] Carlos Lassance, Hervé Déjean, Thibault Formal, and Stéphane Clinchant. 2024. SPLADE-v3: New baselines for SPLADE. *arXiv preprint arXiv:2403.06789* (2024). https://arxiv.org/abs/2403.06789

[21] Carlos Lassance, Simon Lupart, Hervé Déjean, Stéphane Clinchant, and Nicola Tonellotto. 2023. A Static Pruning Study on Sparse Neural Retrievers. *arXiv preprint arXiv:2304.12702* (2023). https://arxiv.org/abs/2304.12702

[22] Xuezhi Li, Zhenmao Zhao, Po Dai, Shuai Zhang, Xinrong Zhang, Zhenman Fang, Liang Wang, and Feng Zhang. 2025. SINDI: An Efficient Index for Sparse Vector Approximate Retrieval. arXiv:2509.08395 doi:10.48550/arXiv.2509.08395

[23] Jimmy Lin and Andrew Trotman. 2015. Anytime Ranking for Impact-Ordered Indexes. In *Proceedings of the 2015 International Conference on the Theory of Information Retrieval (ICTIR)*.

[24] Joel Mackenzie, Andrew Trotman, and Jimmy Lin. 2023. Efficient Document-at-a-Time and Score-at-a-Time Query Evaluation for Learned Sparse Representations. *ACM Transactions on Information Systems* (2023).

[25] Yury A. Malkov and Dmitry A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42, 4 (2020), 824–836.

[26] Antonio Mallia, Omar Khattab, Torsten Suel, and Nicola Tonellotto. 2021. Learning Passage Impacts for Inverted Indexes. In *SIGIR '21: The 44th International ACM SIGIR Conference on Research and Development in Information Retrieval, Virtual Event, Canada, July 11-15, 2021*, Fernando Diaz, Chirag Shah, Torsten Suel, Pablo Castells, Rosie Jones, and Tetsuya Sakai (Eds.). ACM, 1723–1727. doi:10.1145/3404835.3463030

[27] Antonio Mallia, Omar Suel, and Nicola Tonellotto. 2024. Faster Learned Sparse Retrieval with Block-Max Pruning. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '24)*. 2411–2415. doi:10.48550/arXiv.2405.01117

[28] Giuseppe Ottaviano and Rossano Venturini. 2014. Partitioned Elias-Fano Indexes. In *Proceedings of the 37th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*.

[29] Giulio Ermanno Pibiri and Rossano Venturini. 2021. Techniques for Inverted Index Compression. *Comput. Surveys* (2021).

[30] Xinjie Shen, Zhichao Geng, and Yang Yang. 2025. Exploring ℓ_0 Sparsification for Inference-free Sparse Retrievers. In *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '25)*. ACM, 1–5. doi:10.1145/3726302.3730192

[31] Harsha Vardhan Simhadri, Martin Aumüller, Dmitry Baranchuk, Matthijs Douze, Edo Liberty, Amir Ingber, Frank Liu, George Williams, Ben Landrum, Magdalen Dobson Manohar, Neelam Punyani, Shivaram Venkataraman, Ronak Pradeep, Ravishankar Krishnaswamy, Bipin Bhattacharjee, Anshumali Verma, Yibing Ji, Sreenivas Varadhara, Chetan Gupta, Trishul Chilimbi, Mintu Banerjee, and Saurav Ghosh. 2024. Results of the Big ANN: NeurIPS'23 Competition. (2024). arXiv:2409.17424 [cs.IR]

[32] Nandan Thakur, Nils Reimers, Andreas Rücklé, Abhishek Srivastava, and Iryna Gurevych. 2021. BEIR: A Heterogenous Benchmark for Zero-shot Evaluation of Information Retrieval Models. *arXiv preprint arXiv:2104.08663* (4 2021). https://arxiv.org/abs/2104.08663

[33] Howard R. Turtle and James Flood. 1995. Query Evaluation: Strategies and Optimizations. *Information Processing & Management* 31, 6 (1995), 831–850. doi:10.1016/0306-4573(95)00020-H

[34] Xi Zhao, Yihong Chen, Cheng Huang, Yi Zhang, Zhenmao Zhao, and Zhiguang Zheng. 2024. Efficient Approximate Maximum Inner Product Search over Sparse Vectors. In *Proceedings of the 40th IEEE International Conference on Data Engineering (ICDE 2024)*. 3797–3809. doi:10.1109/ICDE60146.2024.00314

[35] Justin Zobel and Alistair Moffat. 2006. Inverted Files for Text Search Engines. *Comput. Surveys* 38, 2 (2006). doi:10.1145/1132956.1132959

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