

# SINDI: An Efficient Index for Sparse Vector Approximate Maximum Inner Product Search

Ruoxuan Li  
ECNU, Shanghai, China  
rxlee@stu.ecnu.edu.cn

Xiaoyao Zhong  
Jiabao Jin  
Ant Group, Shanghai, China  
zhongxiaoyao.zxy@antgroup.com  
jinjiabao.jjb@antgroup.com

Peng Cheng  
Tongji University & ECNU  
Shanghai, China  
cspcheng@tongji.edu.cn

Wangze Ni  
Zhejiang University  
Hangzhou, China  
niwangze@zju.edu.cn

Lei Chen  
HKUST (GZ) & HKUST  
Guangzhou & HK SAR, China  
leichen@cse.ust.hk

Zhitao Shen  
Ant Group, Shanghai, China  
zhitao.szt@antgroup.com

Wei Jia  
Xiangyu Wang  
Ant Group, Shanghai, China  
jw94525@antgroup.com  
wxy407827@antgroup.com

Xuemin Lin  
Shanghai Jiao Tong University  
Shanghai, China  
xuemin.lin@gmail.com

Heng Tao Shen  
Jingkuan Song  
Tongji University, Shanghai, China  
shenhengtao@hotmail.com  
jingkuan.song@gmail.com

## ABSTRACT

Sparse vector Maximum Inner Product Search (MIPS) is crucial in multi-path retrieval for Retrieval-Augmented Generation (RAG). Recent inverted index-based and graph-based algorithms have achieved high search accuracy with practical efficiency. However, their performance in production environments is often limited by redundant distance computations and frequent random memory accesses. Furthermore, the compressed storage format of sparse vectors hinders the use of SIMD acceleration. In this paper, we propose the *sparse inverted non-redundant distance index* (SINDI), which incorporates three key optimizations: (i) Efficient Inner Product Computation: SINDI leverages SIMD acceleration and eliminates redundant identifier lookups, enabling batched inner product computation; (ii) Memory-Friendly Design: SINDI replaces random memory accesses to original vectors with sequential accesses to inverted lists, substantially reducing memory-bound latency. (iii) Vector Pruning: SINDI retains only the high-value non-zero entries of vectors, improving query throughput while maintaining accuracy. We evaluate SINDI on multiple real-world datasets. Experimental results show that SINDI achieves state-of-the-art performance across datasets of varying scales, languages, and models. On the MSMARCO dataset, when Recall@50 exceeds 99%, SINDI delivers single-thread query-per-second (QPS) improvements ranging from 4.2× to 26.4× compared with SEISMIC and PYANNS. Notably, SINDI has been integrated into Ant Group’s open-source vector search library, VSAG.

## PVLDB Reference Format:

Ruoxuan Li, Xiaoyao Zhong, Jiabao Jin, Peng Cheng, Wangze Ni, Lei Chen, Zhitao Shen, Wei Jia, Xiangyu Wang, Xuemin Lin, Heng Tao Shen, and Jingkuan Song. SINDI: An Efficient Index for Sparse Vector Approximate Maximum Inner Product Search. PVLDB, 14(1): XXX-XXX, 2026.  
doi:XX.XX/XXX.XX

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of

## PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/Roxanne0321/vsag/tree/sparse>.

## 1 INTRODUCTION

Recently, retrieval-augmented generation (RAG) [17, 25, 27–29, 31] become one of most successful information retrieval framework attracting attention from research communities and industry. Usually, texts are embedded into dense vectors (i.e., no dimension of the vector is zero entry) in RAG, then retrieved through approximate nearest neighbor search (ANNS) on their corresponding dense vectors.

To enhance the RAG framework, researchers find that using sparse vectors retrieval to complement dense vector based RAG can yield better overall accuracy and recall performance. Different from dense vectors, sparse vectors (i.e., only a very small portion of dimensions of sparse vectors are non-zero entries) are generated by specific models (e.g., SPLADE [7–9, 15]) to preserve semantic information while enabling precise lexical matching [19]. In the enhanced RAG framework, dense vectors capture the holistic semantic similarity between texts and sparse vectors ensure exact term recall, therefore resulting in better overall performance. We show the process of the enhanced RAG in the following example:

**Example 1. Precise lexical matching.** In the retriever stage of RAG, queries and documents are compared to select top-k candidates. Dense vectors capture semantic similarity, while sparse vectors support exact term matching. For example, “I love black cats” is tokenized into “i”, “love”, “black”, and “cats”, with “cats” assigned the highest weight (0.8). A query containing “cats” will

this license. For any use beyond those covered by this license, obtain permission by emailing [info@vldb.org](mailto:info@vldb.org). Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.  
Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.  
doi:XX.XX/XXX.XX

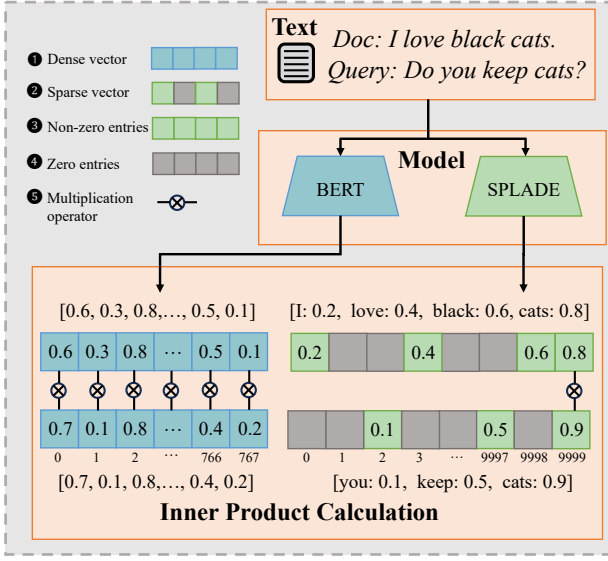


Figure 1: Example of Dense and Sparse Vector Representations and Inner Product Calculations.

precisely match documents where this token has a high weight. **Challenges in inner-product computation.** Dense vectors are stored contiguously, enabling parallel dot-product over consecutive dimensions via SIMD. Sparse vectors typically have very high dimensionality but store only their non-zero entries in a compact format, which leads to two bottlenecks: (1) ID lookup overhead: Matching common non-zero dimensions requires traversing all non-zero entries. Even if only one dimension (e.g., 9999) matches, all entries must be scanned. (2) No SIMD acceleration: Their storage is not dimension-aligned, preventing parallel SIMD processing.

The similarity retrieval problem for sparse vectors is formally known as the Maximum Inner Product Search (MIPS) [13, 24, 30, 32, 34], which aims to identify the top- $k$  vectors in a dataset that have the largest inner product value with a given query vector. However, due to the curse of dimensionality [11], performing exact MIPS in high-dimensional spaces is computationally prohibitive. To mitigate this issue, we focus on Approximate Maximum Inner Product Search (AMIPS) [1, 22, 30], which trades a small amount of recall for significantly improved search efficiency.

Many algorithms [3, 21, 26, 33] have been proposed for AMIPS, employing techniques such as inverted index [3], proximity graphs [26], and hash-based [33] partitioning. They improve efficiency by grouping similar vectors into the same partition, thereby reducing the number of candidates examined during query processing.

Despite reducing the search space, existing approaches still face two major performance bottlenecks: (i) *Distance computation cost*: Matching non-zero dimensions between a query and a document incurs substantial identifier lookup overhead, and the inner product computation cannot be effectively accelerated using SIMD instructions. (ii) *Random memory access cost*: During query processing, data are accessed in a random manner, and the variable lengths of sparse vectors further complicate direct access in memory.

Table 1: Comparison to Existing Algorithms.

	SINDI(ours)	SEISMIC	PYANNS
Distance Complexity	$O\left(\frac{\ q\ }{s}\right)$	$O(\ q\  + \ x\ )$	$O(\ q\  + \ x\ )$
Memory Friendly	✓	✗	✗
SIMD Support	✓	✗	✗
QPS (Recall@50=99%)	241	58	24
Construction Time(s)	58	220	4163

To address the aforementioned challenges, we propose SINDI, a *Sparse Inverted Non-redundant Distance-calculation Index* for efficient sparse vector search. The main contributions of this paper are as follows: (i) *Value-Storing Inverted Index*: SINDI stores both vector identifiers and their corresponding values in the inverted index, enabling direct access during query processing; (ii) *Efficient Inner Product Computation*: SINDI eliminates redundant overhead in identifying common dimensions and fully exploits SIMD acceleration. By grouping values under the same dimension, SINDI enables batched inner product computation during queries; (iii) *Cache-Friendly Design*: SINDI reduces random memory accesses by avoiding fetches of original vectors. Instead, it sequentially accesses inverted lists for specific dimensions, thereby lowering cache miss rates. (iv) *Vector Mass Pruning*: SINDI retains only high-value non-zero entries in vectors, effectively reducing the search space and improving query throughput while preserving accuracy.

We compare SINDI with several state-of-the-art methods on the MSMARCO dataset (8.8M scale) in Table 1. The time complexity of computing the inner product between a query vector  $q$  and a document vector  $x$  using SINDI is  $O\left(\frac{\|q\|}{s}\right)$ , where the involved symbols are defined in Table 2. In contrast, inverted index and graph-based algorithms have a time complexity of  $O(\|q\| + \|x\|)$ . The detailed derivation is given in § 3.2.

In summary, the contributions of this paper are as follows:

- We present SINDI, a novel value-storing inverted index described in §3.1 and §3.2, which reduces redundant distance computation and random memory accesses. We further introduce a *Window Switch* strategy in §3.3 to support large-scale datasets.
- We propose *Vector Mass Pruning* in §4 to decrease the search space and improve query speed while maintaining accuracy.
- We evaluate SINDI on multi-scale, multilingual datasets in §5, demonstrating  $4\times \sim 26\times$  higher single-thread QPS than PYANNS and SEISMIC at over 99% Recall@50, and achieving 8.8M-scale index construction in 60 seconds with minimal cost.

## 2 PRELIMINARIES

### 2.1 Problem Definition

Sparse vectors differ from dense vectors in that most of their dimensions have zero values. By storing only the non-zero entries, they significantly reduce storage and computation costs. We formalize the definition as follows.

**Definition 1** (Sparse Vector and Non-zero Entries). *Let  $\mathcal{D} \subseteq \mathbb{R}^d$  be a dataset of  $d$ -dimensional sparse vectors. For any  $\vec{x} \in \mathcal{D}$ , let  $x$  denote its sparse representation, defined as the set of non-zero entries:*

$$x = \{x^j \mid x^j \neq 0, j \in [0, d-1]\}.$$

**Table 2: Summary of Symbols**

Symbol	Description
$\mathcal{D}$	base dataset
$d$	dimension of $\mathcal{D}$
$\vec{x}, \vec{q}$	base vector, query vector
$x, \ x\ $	sparse format of $\vec{x}$ ; number of non-zero entries in $x$
$\vec{x}_i, x_i$	$i$ -th base vector and its sparse format
$x_i^j$	value of $\vec{x}_i$ in dimension $j$
$s$	SIMD width (elements per SIMD operation)
$\lambda$	window size
$\sigma$	number of windows
$\alpha$	base vector pruning ratio
$\beta$	query vector pruning ratio
$\gamma$	reorder pool size
$I, I_j$	inverted index; inverted list for dimension $j$
$I_{j,w}$	$w$ -th window of inverted list $I_j$
$T^j, T^j[t]$	temporary product array on dimension $j$ ; value at index $t$
$A, A[m]$	distance array; value at index $m$
$\Omega(\vec{x}_1, \vec{x}_2)$	set of common non-zero dimensions of $\vec{x}_1$ and $\vec{x}_2$
$\delta(\vec{x}_1, \vec{x}_2)$	inner product of $\vec{x}_1$ and $\vec{x}_2$

Here,  $x^j$  denotes the value of  $\vec{x}$  in dimension  $j$ . The notation  $\|x\|$  denotes the number of non-zero entries in  $x$ .

To avoid confusion, we illustrate sparse vectors with an example in Figure 1.

**Example 2.** Consider the document “I love black cats” encoded into a sparse embedding:  $[I : 0.2, \text{love} : 0.4, \text{black} : 0.6, \text{cats} : 0.8]$ . The corresponding sparse representation is  $x = \{x^0 = 0.2, x^3 = 0.4, x^{9998} = 0.6, x^{9999} = 0.8\}$ , where  $\|x\| = 4$ .

Since the similarity measure in this work is based on the inner product, we formally define its computation on sparse vectors as follows.

**Definition 2** (Inner Product on Sparse Vectors). Let  $\vec{x}_1, \vec{x}_2 \in \mathcal{D}$ , and let  $x_1$  and  $x_2$  denote their sparse representations. Define the set of common non-zero dimensions as

$$\Omega(\vec{x}_1, \vec{x}_2) = \{j \mid x_1^j \in x_1 \wedge x_2^j \in x_2\}.$$

The inner product between  $\vec{x}_1$  and  $\vec{x}_2$  is then given by

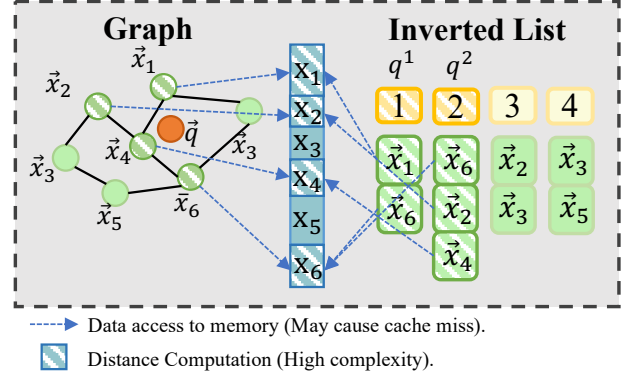
$$\delta(\vec{x}_1, \vec{x}_2) = \sum_{j \in \Omega(\vec{x}_1, \vec{x}_2)} x_1^j \cdot x_2^j.$$

Given the formal definition of the inner product for sparse vectors, we now define the Sparse Maximum Inner Product Search (Sparse-MIPS) task, which aims to find the vector in the dataset that maximizes this similarity measure with the query.

**Definition 3** (Sparse Maximum Inner Product Search). Given a sparse dataset  $\mathcal{D} \subseteq \mathbb{R}^d$  and a query point  $\vec{q} \in \mathbb{R}^d$ , the Sparse Maximum Inner Product Search (Sparse-MIPS) returns a vector  $\vec{x}^* \in \mathcal{D}$  that has the maximum inner product with  $\vec{q}$ , i.e.,

$$\vec{x}^* = \arg \max_{\vec{x} \in \mathcal{D}} \delta(\vec{x}, \vec{q}). \quad (1)$$

For small datasets, exact Sparse-MIPS can be obtained by scanning all vectors. For large-scale high-dimensional collections, this is prohibitively expensive, and Approximate Sparse-MIPS mitigates the cost by trading a small loss in accuracy for much higher efficiency.



**Figure 2: The Bottleneck of the Graph Index and Inverted Index During Searching Process.**

**Definition 4** (Approximate Sparse Maximum Inner Product Search). Given a sparse dataset  $\mathcal{D} \subseteq \mathbb{R}^d$ , a query point  $\vec{q}$ , and an approximation ratio  $c \in (0, 1]$ , let  $\vec{x}^* \in \mathcal{D}$  be the vector that has the maximum inner product with  $\vec{q}$ . A  $c$ -maximum inner product search ( $c$ -MIPS) returns a point  $\vec{x} \in \mathcal{D}$  satisfying  $\delta(\vec{q}, \vec{x}) \geq c \cdot \delta(\vec{q}, \vec{x}^*)$ .

In practice,  $c$ -Sparse-MIPS methods can reduce query latency by orders of magnitude compared with exact search, making them preferable for large-scale, real-time applications such as web search, recommender systems, and computational advertising.

For ease of reference, the main notations and their meanings are summarized in Table 2, which will be referred to throughout the rest of the paper.

## 2.2 Existing Solutions

Representative algorithms for the AMIPS problem on sparse vectors include the inverted-index based SEISMIC [3], the graph based PYANNS [26]. SEISMIC constructs an inverted list based on vector dimensions. PYANNS creates a proximity graph where similar vectors are connected as neighbors.

**Example 3.** Figure 2 illustrates a proximity graph and an inverted index constructed for  $\vec{x}_1$  to  $\vec{x}_6$ . Consider a query vector  $\vec{q}$  with two non-zero entries  $q^1$  and  $q^2$ . In the proximity graph, when the search reaches  $\vec{x}_4$ , the algorithm computes distances between  $\vec{q}$  and all its neighbors, sequentially accessing  $x_1, x_2, x_4$ , and  $x_6$  from memory. In the inverted index, the algorithm traverses the posting lists for dimensions 1 and 2, accessing  $x_1, x_6, x_2$ , and  $x_4$ . Since vector access during search is essentially random, this incurs substantial random memory access overhead. Moreover, because  $\|x\|$  varies across vectors, the distance computation between  $\vec{q}$  and  $\vec{x}$  has a time complexity of  $O(\|q\| + \|x\|)$ .

**Redundant Distance Computations.** Sparse vectors incur high distance computation cost due to (i) the explicit lookup needed to identify the common dimensions  $\Omega(\vec{x}, \vec{q})$  between a document  $\vec{x}$  and a query  $\vec{q}$ , resulting in complexity  $O(\|q\| + \|x\|)$ , and (ii) the inability of existing algorithms to exploit SIMD acceleration for inner product computation. Profiling 6980 queries on the MSMARCO dataset (1M vectors) using perf [18] and VTune [12] shows that PYANNS spent 83.3% of CPU cycles on distance calculation.

**Random Memory Accesses.** The inefficiency of memory access in existing algorithms can be attributed to two main factors. First, SEISMIC organizes similar data points into the same partition. To improve accuracy, vectors are replicated across multiple partitions. This replication breaks the alignment between storage layout and query traversal order, preventing cache-friendly sequential access. During retrieval, the index returns candidate vector IDs, which incur random memory accesses to fetch their corresponding data, leading to frequent cache misses. Moreover,  $\|x\|$  varies across sparse vectors, requiring offset table lookups to locate each vector’s data. In our measurements, SEISMIC averaged 5168 random vector accesses per query (5.1 MB), with an L3 cache miss rate of 67.68%.

### 3 FULL PRECISION INVERTED INDEX

This section introduces full-precision SINDI, an inverted index designed for sparse vector retrieval. Its advantages are organized along three aspects: index structure, distance computation, and cache optimization.

- In § 3.1, SINDI constructs a value-based inverted index by storing both vector identifiers and their corresponding dimension values in posting lists. This design eliminates the redundant dimension-matching overhead present in traditional inverted indexes.
- In § 3.2, SINDI employs a two-phase search process involving *product computation* and *accumulation*. By using SIMD instructions in multiplication, it reduces query complexity from  $O(\|q\| + \|x\|)$  to  $O\left(\frac{\|q\|}{s}\right)$ . This maximizes CPU utilization and improves query throughput.
- In § 3.3, to mitigate cache misses caused by random access to the distance array, SINDI introduces *Window Switch* strategy, which partitions each posting list into fixed-size segments of length  $\lambda$  while using a shared distance array. This reduces memory overhead without increasing computation, and both theoretical analysis and experimental evaluation (Figure 5) demonstrate the existence of an optimal  $\lambda$  that minimizes memory access overhead.

#### 3.1 Value-storing Inverted Index

Redundant inner product computations arise because identifying the common non-zero dimensions  $\Omega(\vec{q}, \vec{x})$  between a query vector  $\vec{q}$  and a document vector  $\vec{x}$  requires scanning many irrelevant entries outside their intersection. We observe that the document identifiers retrieved from traversing an inverted list correspond precisely to the dimensions in  $\Omega(\vec{x}, \vec{q})$ . Therefore, when accessing a document  $\vec{x}$  from the list of dimension  $j$ , we can simultaneously retrieve its value  $x_j$ , thereby enabling direct computation of the inner product without incurring the overhead of finding  $\Omega(\vec{q}, \vec{x})$ .

**Example 4.** Figure 3 illustrates the inverted lists constructed for vectors  $x_1$  to  $x_5$ . When a query  $q$  arrives, it sequentially probes the inverted lists for dimensions 1, 3, and 5. In the base dataset, only the dimensions belonging to  $\Omega(\vec{q}, \vec{x})$  need to be traversed during the inner product computation. For example, although  $x_4$  has a value in dimension 2, this dimension is not in  $\Omega(\vec{q}, x_4)$  and thus is never accessed during the computation. We further observe that the document identifiers retrieved from the inverted lists overlap exactly with those used to determine the common non-zero dimensions for the inner product. This implies that the products of these non-zero

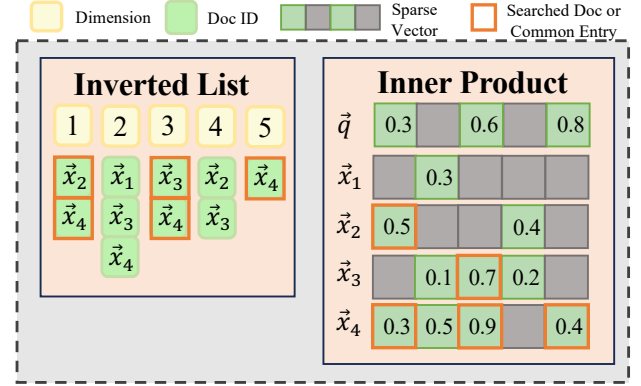


Figure 3: Overlap of Inverted List Entries and Common Non-Zero Dimensions in Inner-Product Computation.

entries can be computed during the document retrieval process itself, thereby ensuring that only dimensions in  $\Omega(\vec{q}, \vec{x})$  are involved in the inner product computation.

Inspired by these observations, we extend each inverted list  $I_j$  to store not only the identifier  $i$  of the vector  $\vec{x}_i$ , but also the value  $x_i^j$  in dimension  $j$ . In our notation, we simply write  $x_i^j$  in  $I_j$  to denote this stored value, with the subscript  $i$  implicitly encoding the associated vector ID. This value-storing design eliminates the cost of explicitly locating  $\Omega(\vec{x}_i, \vec{q})$  during inner product computation, as well as the additional random memory access that would otherwise be required to fetch  $x_i^j$  from the original vector.

#### 3.2 Efficient Distance Computation

With the value-storing inverted index, the inner product between a query  $\vec{q}$  and candidate vector  $\vec{x}$  can be computed without explicitly identifying  $\Omega(\vec{q}, \vec{x})$ . During search, SINDI organizes the computation into two stages: *product computation* and *accumulation*.

In the first stage, as each relevant posting list  $I_j$  is traversed, the products  $q^j \times x_i^j$  are computed (using SIMD instructions when possible) and stored sequentially in a temporary array  $T^j$ . Here,  $T^j$  holds the products for  $I_j$ , with each entry  $T^j[t]$  corresponding to the  $t$ -th entry in  $I_j$ . Therefore,  $T^j$  has the same length as  $I_j$ .

In the second stage, the values in  $T^j$  are aggregated into a pre-allocated distance array  $A$  of length  $\|\mathcal{D}\|$ , where each entry  $A[i]$  stores the accumulated score for vector  $\vec{x}_i$ . This arrangement enables  $O(1)$  time per accumulation into  $A$ , and naturally exploits SIMD parallelism.

The following example illustrates the detailed steps of the search procedure introduced above.

**Example 5.** Figure 4 illustrates the search procedure. Since  $\|\mathcal{D}\| = 9$ , the distance array  $A$  is initialized with  $\text{size}(A) = 9$  and all elements set to 0. The query  $\vec{q}$  contains three non-zero components,  $q^1$ ,  $q^5$ , and  $q^8$ , so only the inverted lists  $I_1$ ,  $I_5$ , and  $I_8$  are traversed. Consider  $\vec{x}_4$  as an example. From  $I_1$ , we obtain  $x_4^1 = 6.8$ , and the product  $q^1 \times x_4^1 = 17.0$  (this multiplication can be SIMD-accelerated) is temporarily stored in  $T[0]$ . Accumulating  $T[0]$  into  $A[4]$  gives  $A[4] = 0 + 17.0 = 17.0$ . Similarly, from  $I_5$  we compute  $x_4^5 \times q^5 = 14.0$ , store it in  $T[2]$ , and add it to obtain  $A[4] = 31.0$ . The computation for  $I_8$  proceeds analogously. Finally,  $A[4]$  becomes 36.1, which

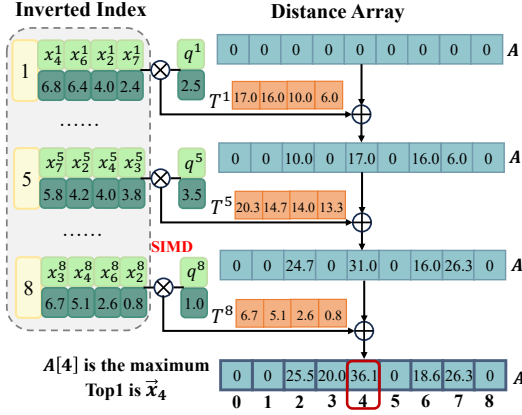


Figure 4: An Example of SINDI and Query Process.

equals  $\delta(\vec{x}_4, \vec{q})$ . Although  $\vec{x}_4$  has another non-zero entry  $x_4^2$ , it is not in  $\Omega(\vec{x}_4, \vec{q})$  and thus does not contribute to the inner product. The same accumulation process applies to  $\vec{x}_2$ ,  $\vec{x}_3$ ,  $\vec{x}_6$ , and  $\vec{x}_7$ . Eventually,  $A[4]$  holds the largest value, so the nearest neighbor of  $\vec{q}$  is  $\vec{x}_4$ .

Using SIMD instructions, SINDI processes the  $j$ -th inverted list  $I_j$  in batches, multiplying  $q^j$  with each  $x_i^j$  it contains and writing the results sequentially into  $T^j$ . This approach maximizes CPU utilization and reduces the complexity of the inner product computation from  $O(\|q\| + \|x\|)$  to  $O(\frac{\|q\|}{s})$ , where  $s$  denotes the number of elements processed per SIMD operation. The complexity of SINDI's distance computation is derived as follows:

**Theorem 3.1 (Amortized Time Complexity of SINDI Distance Computation).** Let  $\mathcal{D}$  be the dataset,  $I$  its inverted index, and  $I_j$  the posting list for dimension  $j$ . Let  $\mathcal{I}^j = \{x_i^j \mid x_i^j \neq 0\}$  denote the set of non-zero entries in  $I_j$ .

Given a query vector  $\vec{q}$ , let  $\mathcal{J} = \{j \mid q^j \neq 0\}$  be the set of query dimensions with non-zero entries. Let  $\mathcal{X} = \{\vec{x}_i \mid \exists j \in \mathcal{J}, x_i^j \neq 0\}$  denote the set of candidate vectors retrieved by  $\vec{q}$ .

Let  $s$  be the number of elements that can be processed simultaneously using SIMD instructions. Then the amortized per-vector time complexity of computing the inner product between  $\vec{q}$  and all  $\vec{x}_i \in \mathcal{X}$  is  $\Theta(\frac{\|q\|}{s})$ .

**Proof.** The total number of non-zero entries accessed in all posting lists for  $\mathcal{J}$  is

$$\sum_{j \in \mathcal{J}} \|\mathcal{I}^j\|.$$

Since  $s$  entries can be processed in parallel using SIMD, the total time is

$$T_{\text{total}} = \sum_{j \in \mathcal{J}} \frac{\|\mathcal{I}^j\|}{s}.$$

Amortizing over all  $\|\mathcal{X}\|$  candidates gives

$$T = \frac{T_{\text{total}}}{\|\mathcal{X}\|} = \frac{\sum_{j \in \mathcal{J}} \|\mathcal{I}^j\|}{s \cdot \|\mathcal{X}\|}.$$

For any  $\vec{x}_i \in \mathcal{X}$ , we have  $q \cap x_i \subseteq q$ , implying that  $\|\Omega(\vec{x}_i, \vec{q})\| \leq \|q\|$ . Therefore:

$$T \leq \frac{\sum_{\vec{x}_i \in \mathcal{X}} \|q\|}{s \cdot \|\mathcal{X}\|} = \frac{\|q\| \cdot \|\mathcal{X}\|}{s \cdot \|\mathcal{X}\|} = \frac{\|q\|}{s}.$$

Substituting into  $T$  yields

$$T \leq \frac{\sum_{\vec{x}_i \in \mathcal{X}} \|q\|}{s \cdot \|\mathcal{X}\|} = \frac{\|q\|}{s}.$$

Hence, the amortized per-vector complexity is

$$\Theta\left(\frac{\|q\|}{s}\right).$$

□

In summary, the SINDI index removes the overhead of redundant dimension matching and exploits SIMD parallelism to compute inner products in batches, thereby fully utilizing CPU computational resources. It also eliminates the memory access costs of traversing the original vectors, as the required values are retrieved directly from the posting lists and used for in-place product computation during retrieval.

### 3.3 Cache Optimization

When the dataset size  $\mathcal{D}$  reaches the million scale, the distance array  $A$  becomes correspondingly large. Random accesses to such a long array cause frequent cache misses, making query performance highly memory-bound. In addition, allocating a full-length distance array for every query incurs substantial memory overhead.

To address these issues, SINDI adopts the *Window Switch* strategy, which partitions the vector ID space into fixed-size windows and restricts each query to accessing IDs within a single window at a time. With a shorter distance array per window, the accessed entries are located within a much more compact memory region. This substantially improves spatial locality, and the resulting access pattern closely resembles a sequential scan, enabling hardware prefetching and reducing cache misses.

**3.3.1 Window Switch.** During index construction, SINDI partitions the dataset  $\mathcal{D}$  into contiguous ID segments, referred to as *windows*. The window size is denoted by  $\lambda$  ( $0 < \lambda \leq \|\mathcal{D}\|$ ), and the total number of windows is  $\sigma = \lceil \frac{\|\mathcal{D}\|}{\lambda} \rceil$ . The  $w$ -th window contains vectors from  $\vec{x}_{w\lambda}$  to  $\vec{x}_{(w+1)\lambda-1}$ , and the window index of vector  $\vec{x}_i$  is  $\lfloor \frac{i}{\lambda} \rfloor$ . Each inverted list  $I_j$  is partitioned in the same way, so every list has  $\sigma$  windows. We denote the  $w$ -th window of the  $j$ -th inverted list by  $I_{j,w}$  ( $0 \leq w < \sigma$ ), whose entries are the non-zero  $x_i^j$  in dimension  $j$  for vectors in that ID range. Each  $x_i^j$  implicitly carries the identifier  $i$  through its subscript.

At query time, the length of the distance array  $A$  is set to the window size  $\lambda$ , and all windows share this same  $A$ . Within a window, each vector  $\vec{x}_i$  is mapped to a unique entry  $A[i \bmod \lambda]$ .

The search procedure for the  $w$ -th window proceeds in two steps:

**(1) Inner product computation.** For each scanned posting list  $I_{j,w}$ , compute the products  $q^j \times x_i^j$  for all its entries, using SIMD instructions when possible. These products are written sequentially into a temporary array  $T^j$ , which has the same length as  $I_{j,w}$  and is index-aligned with it — i.e.,  $T^j[t]$  stores the product for the  $t$ -th posting in  $I_{j,w}$ . The accumulation stage then adds each  $T^j[t]$  into the corresponding  $A[i \bmod \lambda]$  using  $O(1)$  time per update.

**(2) Heap update.** After processing all query dimensions for the current window,  $A$  holds the final scores for that window's candidates. Scan  $A$  to insert the top-scoring vectors into a minimum heap  $H$ , which maintains the vector IDs and scores for the results to be



---

**Algorithm 1: PRECISESINDICONSTRUCTION**

---

**Input:** A sparse dataset  $\mathcal{D}$  and dimension  $d$ , window size  $\lambda$   
**Output:** Inverted list  $I$

```
1 for  $j \in \{0, \dots, d-1\}$  do
2    $\mathcal{X} \leftarrow \{\vec{x}_i \in \mathcal{D} \mid x_i^j \neq 0\}$ ;
3   foreach  $\vec{x}_i \in \mathcal{X}$  do
4      $w \leftarrow \lfloor \frac{i}{\lambda} \rfloor$ 
5      $I_{j,w}.append(x_i^j)$ 
6 return  $I$ 
```

---

returned. Here, each  $A[t]$  corresponds to vector  $\vec{x}_{t+\lambda \times w}$ , so the global ID can be recovered from the local index  $t$ .

Note that *Window Switch* only changes the order of list entries scanned, without altering the number of arithmetic operations. Thus, the time complexity of distance computation remains  $O\left(\frac{\|q\|}{s}\right)$ .

**3.3.2 Construction and Search.** The construction process of the full-precision SINDI index with *Window Switch* is shown in Algorithm 1. Given a sparse vector dataset  $\mathcal{D}$  of dimension  $d$ , the algorithm iterates over each dimension  $j$  (Line 1). For each  $j$ , it collects all vectors  $\vec{x}_i \in \mathcal{D}$  having a non-zero entry  $x_i^j$  into a temporary set  $\mathcal{X}$  (Line 2). These vectors are then appended to the corresponding window  $I_{j,w}$  of  $I_j$  based on their IDs, where  $w = \lfloor i/\lambda \rfloor$  (Lines 3–5). After processing all dimensions, the inverted index  $I$  is returned (Line 6). The time complexity of the construction process is  $O(\|\mathcal{D}\| \|\vec{x}\|)$ , where  $\|\vec{x}\| = \frac{\sum_{\vec{x}_i \in \mathcal{D}} \|\vec{x}_i\|}{\|\mathcal{D}\|}$  denotes the average number of nonzero entries per vector.

The search process for the full-precision SINDI index is summarized in Algorithm 2, and consists of three stages: product computation, accumulation, and heap update. Given a query  $\vec{q}$ , inverted index  $I$ , and target recall size  $k$ , a distance array  $A$  of length  $\lambda$  is initialized to zeros (Lines 1–2) and an empty min-heap  $H$  is created (Line 3). The outer loop traverses all windows  $w \in \{0, \dots, \sigma-1\}$  (Line 4). For each non-zero query component  $q^j$  (Line 5), SIMD-based batched multiplication is performed with all  $x_i^j$  in  $I_{j,w}$ , and the results are stored sequentially into a temporary product array  $T^j$  aligned with  $I_{j,w}$  (Line 6). Each  $x_i^j$  is then retrieved (Lines 7–8), its mapped index  $m = i \bmod \lambda$  computed (Line 9), and  $T^j[t]$  accumulated into  $A[m]$  (Line 10). After all  $q^j$  for the current window are processed, the heap update stage begins (Line 12): each  $A[m]$  that exceeds the heap minimum or when  $H$  has fewer than  $k$  entries is inserted into  $H$  with its global ID  $(m + \lambda w)$  and score  $A[m]$  (Lines 13–14), removing the smallest if size exceeds  $k$  (Lines 15–16).  $A[m]$  is then reset to zero for the next window (Line 17). When all windows are processed,  $H$  contains up to  $k$  vector IDs with their full-precision distances to  $\vec{q}$ , which are returned as the final result (Line 20).

**Complexity.** Let  $l = \frac{\sum_{q^j \in q} |I_j|}{\|q\|}$  denote the average number of nonzero entries in the traversed lists. With *Window Switch*, the total number of postings visited is still  $\|q\| l$ , so the time complexity of a full-precision query is  $O\left(\frac{\|q\| l}{s}\right)$ , where  $s$  is the SIMD batch size. Hence, the computational cost is independent of the window size  $\lambda$ .

---

**Algorithm 2: PRECISESINDISEARCH**

---

**Input:** Query  $\vec{q}$ , an inverted list  $I$ , and  $k$   
**Output:** At most  $k$  points in  $\mathcal{D}$

```
1 for  $m \in \{0, \dots, \lambda-1\}$  do
2    $A[m] \leftarrow 0$ 
3  $H \leftarrow$  empty min-heap
4 for  $w \in \{0, \dots, \sigma-1\}$  do
5   foreach  $q^j \in q$  do
6      $T^j \leftarrow \text{SIMDProduct}(q^j, I_{j,w})$ ;
7     for  $t \in \{0, \dots, I_{j,w}.size()-1\}$  do
8        $x_i^j \leftarrow I_{j,w}[t]$ ;
9        $m \leftarrow i \bmod \lambda$ ;
10       $A[m] \leftarrow A[m] + T^j[t]$ ;
11   for  $m \in \{0, \dots, \lambda-1\}$  do
12     if  $A[m] > H.min()$  or  $H.len() < k$  then
13        $H.insert(m + \lambda \times w, A[m])$ 
14     if  $H.len() > k$  then
15        $H.pop()$ 
16    $A[m] \leftarrow 0$ 
17 return  $H$ 
```

---

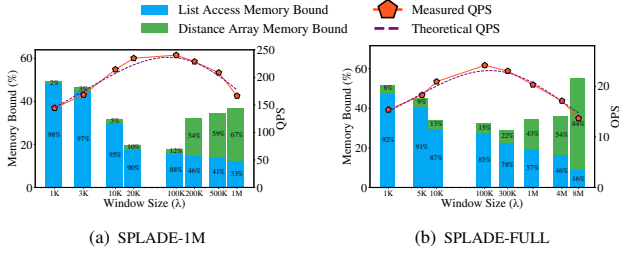
### 3.4 Analysis of Window Size’s Impact on Performance

While the *Window Switch* strategy does not alter the overall computational complexity, two types of memory-access costs are sensitive to the window size  $\lambda$ :

- **Random-access cost to the distance array.** During accumulation, each partial score  $T^j[t]$  is added to  $A[i \bmod \lambda]$ , producing a random write pattern over  $A$ . When  $\lambda$  decreases, the length of  $A$  becomes smaller, more of it fits in cache, and this random-access cost decreases due to fewer cache misses.
- **Cache eviction to sub-list cost when switching windows.** Under *Window Switch*, the search iterates over multiple posting sub-lists  $I_{j,w}$  for different dimensions  $j$  within the same window  $w$ . Switching from one dimension’s sub-list to another may evict previously cached sub-list data from memory. When  $\lambda$  decreases, the number of windows  $\sigma = \frac{\|\mathcal{D}\|}{\lambda}$  increases, leading to more frequent loading and eviction of these sub-lists, and hence increasing this cost.

Selecting  $\lambda$  thus requires balancing the reduced random-access cost to the distance array against the increased cache-eviction cost to inverted sub-lists. The following example illustrates this trade-off:

**Example 6.** Figure 5 reports experimental results for the full-precision SINDI on the SPLADE-1M and SPLADE-FULL datasets. For each dataset, we executed 6,980 queries under different window sizes  $\lambda$  and measured the QPS. We also used the Intel VTune Profiler [12] to record memory bound metrics for two types of memory accesses: random accesses to the distance array (arising from accumulation writes) and cache eviction to sub-lists (occurring when switching dimensions within a window). Here, memory bound denotes the percentage of execution time stalled due to memory accesses. For the SPLADE-1M dataset, as  $\lambda$  increases from 1K to 1M, the memory bound from distance array accesses increases



**Figure 5: Impact of Window Size on Query Throughput and Memory Accesses.**

monotonically, whereas that from sub-list cache evictions decreases monotonically. The total memory-bound latency reaches its minimum near  $\lambda \approx 100K$ , corresponding to the highest query throughput. The SPLADE-FULL dataset exhibits the same trend, confirming the existence of an optimal window size.

Based on this, the memory-access latency for queries can be modeled by a double power-law [2, 10]:

$$T_{\text{mem}}(\lambda) = A\lambda^{\alpha} + B\lambda^{-\beta} + C, \quad (2)$$

where:

- $\lambda$  is the window size;
- $A\lambda^{\alpha}$  models the increasing cost of random accesses to the distance array as  $\lambda$  grows;
- $B\lambda^{-\beta}$  models the decreasing cost of cache eviction to sub-lists with larger  $\lambda$ ;
- $C$  is the baseline memory-access cost unrelated to  $\lambda$ .

This function reaches its minimum at  $\lambda^* = \left(\frac{B\beta}{A\alpha}\right)^{\frac{1}{\alpha+\beta}}$ . When  $\lambda \ll \lambda^*$ ,  $T_{\text{mem}}$  is dominated by the sub-list eviction term and decreases as  $\lambda$  increases; when  $\lambda \gg \lambda^*$ , the distance-array term dominates and  $T_{\text{mem}}$  grows with  $\lambda$ . The optimum  $\lambda^*$  occurs when these two terms are balanced.

**Example 7.** Figure 5 shows that the dashed line represents the theoretical QPS curve. It is obtained by estimating  $(\alpha, \beta)$  via log-log regression [6] and  $(A, B, C)$  via least-squares fitting [16] of the double power-law model. For the SPLADE-1M dataset, the model predicts an optimal  $\lambda^* \approx 7.35 \times 10^4$ , while for SPLADE-FULL it predicts  $\lambda^* \approx 1.25 \times 10^5$ . These predictions are reasonably consistent with the measured optimal  $\lambda \approx 10^5$ , with minor discrepancies attributable to hardware-level performance variability and measurement noise introduced by VTune profiling.

## 4 APPROXIMATE INVERTED INDEX

This section focuses on optimizing the query process through pruning and reordering. In our framework, pruning serves as a form of coarse retrieval, reducing the number of non-zero entries or inverted-list lengths to quickly generate a compact candidate set. *Reordering* then plays the role of fine-grained ranking by computing exact inner products for this candidate set. Combining these two stages significantly improves query throughput while keeping the loss in recall negligible.

### 4.1 Pruning Strategies

A key property of sparse vectors is that a small number of high-valued non-zero entries can preserve the majority of the vector’s overall information content [9]. This distribution pattern typically results from the training objectives of sparse representation models. For instance, SPLADE often concentrates the most informative components into a limited set of high-weight dimensions. Conversely, many low-valued non-zero entries correspond to common stopwords (e.g., “is”, “the”), which can be safely removed with minimal impact on retrieval quality.

Let  $\tilde{x}_i$  denote the pruned version of document  $\tilde{x}_i$ , and  $\tilde{q}'$  the pruned version of query  $\tilde{q}$ . Let  $l$  be the average posting-list length before pruning and  $l'$  the average posting-list length after pruning. The reduction in computational cost achieved by pruning is

$$\|q\| \cdot l - \|q'\| \cdot l',$$

For a given pruning operator  $\phi$ , we define the *inner product error* for document  $\tilde{x}_i$  as

$$e_i^{(\phi)} = \delta(\tilde{x}_i, \tilde{q}) - \delta(\phi(\tilde{x}_i), \phi(\tilde{q})),$$

where  $\delta(\cdot, \cdot)$  denotes the exact inner product and  $\phi(\cdot)$  applies the pruning transformation. The total inner product error over the dataset  $\mathcal{D}$  is then

$$\varepsilon^{(\phi)} = \sum_{\tilde{x}_i \in \mathcal{D}} e_i^{(\phi)}$$

Smaller  $\|x'_i\|$  yields higher query throughput, typically incurs a larger inner product error. Therefore, pruning must be designed with a trade-off between efficiency and accuracy. The following example shows that it is possible to retain a subset of high-value non-zero entries while incurring merely a small loss in inner product accuracy.

**Example 8.** Figure 6(a) reports the inner product error measured on a 100K-scale dataset when retaining different proportions of the largest non-zero entries from both document and query vectors. The results show that the error drops rapidly as the retaining ratio increases from 0.1 to 0.3, and becomes almost negligible once the ratio exceeds 0.5. This indicates a saturation effect, where further increasing the retaining ratio yields minimal additional gains in accuracy.

As discussed in Section 3, for full-precision SINDI the upper bound for computing the inner product between  $\tilde{x}_i$  and  $\tilde{q}$  is  $\Theta\left(\frac{\|q\|}{s}\right)$ .

For a given  $\tilde{q}$ , the overall query complexity is  $\mathcal{O}\left(\frac{\|q\|l}{s}\right)$ , where  $l$  is the average number of inverted lists traversed. Thus, reducing query latency requires decreasing  $l$ ,  $\|x\|$ , and  $\|q\|$ . This can be achieved via three approaches: list pruning, document pruning, and query pruning. List and document pruning are performed during index construction, while query pruning is applied at query time. In this work, we focus on the construction stage, since query pruning and document pruning are both forms of vector pruning. We compare three strategies—*List Pruning (LP)*, *Vector Number Pruning (VNP)*, and *Mass Ratio Pruning (MRP)*—and analyze their respective strengths and weaknesses.

**List Pruning (LP).** LP operates at the inverted-list level: for each dimension  $j$ , it retains only the non-zero entries with the largest absolute values in  $I_j$ , limiting the list length to  $l'$ . Since the size of  $I_j$  varies across dimensions, some high-value  $|x'_i|^j$  entries in longer

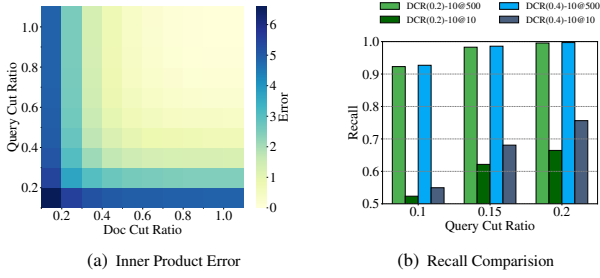


Figure 6: Intuition of Pruning and Reorder

lists may be removed, while lower-value entries in shorter lists may be kept. After pruning, each document vector  $\vec{x}_i$  becomes  $\phi_{LP}(\vec{x}_i)$ , containing only the coordinates that survive the list truncation.

**Vector Number Pruning (VNP).** VNP applies the pruning operator  $\phi_{VNP}$  at the vector level. For each document vector  $\vec{x}_i$ ,  $\phi_{VNP}(\vec{x}_i)$  retains the  $vn$  non-zero entries with the largest absolute values, ensuring  $\|\phi_{VNP}(\vec{x}_i)\| = vn$ . Since  $\|\vec{x}_i\|$  varies across vectors, high-value  $|x_i^j|$  entries that contribute substantially to the inner product may still be removed under this fixed-size scheme.

**Mass Ratio Pruning (MRP).** MRP applies the pruning operator  $\phi_{MRP}$  based on the cumulative sum of the absolute values of a vector’s non-zero entries. For each document vector  $\vec{x}_i$ ,  $\phi_{MRP}(\vec{x}_i)$  ranks all non-zero entries in descending order of absolute value and retains the shortest prefix whose cumulative sum reaches a fraction  $\alpha$  of the vector’s total mass. This adaptive scheme removes low-value entries that contribute little to the inner product, while allowing vectors with different value distributions to keep variable numbers of entries, thereby reducing inverted-list size without enforcing a uniform length limit.

To formally introduce MRP, we first define the mass of a vector.

**Definition 5 (Mass of a Vector).** Let  $\vec{x} \in \mathbb{R}^d$  be a vector. The *mass* of  $\vec{x}$  is defined as the sum of the absolute values of  $x$ ’s non-zero entries:

$$mass(\vec{x}) = \sum_{x^j \neq 0} |x^j|.$$

We define the vector obtained after Mass Ratio Pruning as the  $\alpha$ -mass subvector.

**Definition 6 ( $\alpha$ -Mass Subvector).** Let  $\vec{x} \in \mathbb{R}^d$  and let  $\pi$  be a permutation that orders the non-zero entries of  $\vec{x}$  by non-increasing absolute value, i.e.,  $|x^{\pi_j}| \geq |x^{\pi_{j+1}}|$ . For a constant  $\alpha \in (0, 1]$ , let  $1 \leq r \leq \|\vec{x}\|$  be the smallest integer satisfying

$$\sum_{j=1}^r |x^{\pi_j}| \geq \alpha \times mass(\vec{x}).$$

The  $\alpha$ -mass subvector, denoted  $\alpha\text{-mass}(\vec{x})$ , is the vector whose non-zero entries are  $\{x^{\pi_j}\}_{j=1}^r$ .

**Example 9.** Figure 7 illustrates the three pruning methods applied to sparse vectors  $\vec{x}_1$ ,  $\vec{x}_2$ , and  $\vec{x}_3$ . List Pruning prunes each inverted list to size  $l' = 2$ , Vector Number Pruning retains the top  $vn = 2$  entries of each vector, and Mass Ratio Pruning prunes each  $\vec{x}_i$  to its  $\alpha\text{-mass}(\vec{x}_i)$  with  $\alpha = 0.7$ . The figure shows: (i) all three strategies result in the same reduction in computation,  $\|q\|l - \|q'\|l' =$

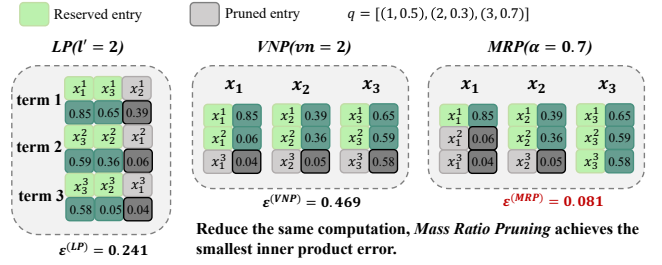


Figure 7: An Example of List Pruning, Vector Number Pruning and Mass Ratio Pruning.

---

#### Algorithm 3: APPROXIMATE SINDI CONSTRUCTION

---

**Input:** Sparse dataset  $\mathcal{D}$  of dimension  $d$ ; window size  $\lambda$ ; pruning ratio  $\alpha$

**Output:** Inverted index  $I$

```

1  $\mathcal{D}' \leftarrow \emptyset$ 
2 foreach  $\vec{x}_i \in \mathcal{D}$  do
3    $\vec{x}'_i \leftarrow \alpha\text{-mass}(\vec{x}_i)$ 
4    $\mathcal{D}' \leftarrow \mathcal{D}' \cup \vec{x}'_i$ 
5  $I = \text{PRECISE SINDI CONSTRUCTION}(\mathcal{D}', d, \lambda)$ 
6 return  $I$  and  $\mathcal{D}$ 
```

---

$9 - 6 = 3$ ; (ii) Mass Ratio Pruning yields the smallest inner product error. This is because List Pruning cannot retain the larger value  $x_2^1$  when each list is limited to two vectors, and Vector Number Pruning drops  $x_3^3$ . In contrast, Mass Ratio Pruning prioritizes high-value entries, thereby minimizing error.

Algorithm 3 outlines the construction of the approximate SINDI index. Given a sparse dataset  $\mathcal{D}$  of maximum dimension  $d$ , window size  $\lambda$ , and pruning ratio  $\alpha$ , the algorithm first initializes an empty set  $\mathcal{D}'$  to store pruned vectors (Line 1). For each vector  $\vec{x}_i \in \mathcal{D}$  (Line 2), its  $\alpha$ -mass subvector  $\alpha\text{-mass}(\vec{x}_i)$  is computed and assigned to  $\vec{x}'_i$  (Line 3), which is then added to  $\mathcal{D}'$ . The remaining steps invoke PRECISE SINDI CONSTRUCTION on  $\mathcal{D}'$  (Line 5), followed by returning both the inverted index  $I$  and the original dataset  $\mathcal{D}$  (Line 6) so that reordering can be performed during retrieval.

Figure 6(a) shows that retaining under half of a query’s non-zero entries reduces the inner product error to nearly zero, cutting the search space by more than half. This suggests that posting lists from a few high-value non-zero entries in a query already cover most of the recall. Therefore, SINDI applies Mass Ratio Pruning to queries: given  $\beta \in (0, 1]$ , the pruned query is denoted  $\beta\text{-mass}(\vec{q})$  and is used in coarse retrieval.

## 4.2 Reordering

Retaining only a small portion of non-zero entries preserves most of the inner product but may disrupt the partial order of full inner products. Using such pruned results directly for recall degrades accuracy. Nevertheless, experiments show that with enough candidates, the true nearest neighbors are often included. Figure 6(b) reports Recall 10@500 and Recall 10@10 under different pruning ratios for documents and queries. Retaining 20% of document entries and 15% of query entries yields Recall 10@500 = 0.98 but Recall 10@10 = 0.63. This motivates a two-step strategy: (1) perform coarse recall with the



---

**Algorithm 4:** APPROXIMATE SINDISEARCH

---

**Input:** Query  $\vec{q}$ , inverted index  $I$ , query prune ratio  $\beta$ , reorder number  $\gamma$ , and  $k$   
**Output:** Top- $k$  points in  $\mathcal{D}$  (at most  $k$ )

```
1  $H \leftarrow$  empty min-heap
2  $R \leftarrow$  empty max-heap
3  $\vec{q}' \leftarrow \beta\text{-mass}(\vec{q})$ 
4  $H \leftarrow \text{PRECISESINDISEARCH}(\vec{q}', I, \gamma)$ 
5 while  $H \neq \emptyset$  do
6    $i, dis \leftarrow H.pop()$ 
7    $dis' \leftarrow 1 - \delta(\vec{x}_i, \vec{q})$ 
8   if  $dis' < R.max()$  or  $R.len() < k$  then
9      $R.insert(i, dis')$ 
10  if  $R.len() > k$  then
11     $R.pop()$ 
12 return  $R$ 
```

---

pruned index to retrieve  $\gamma$  candidates into a min-heap  $H$ ; (2) compute the full inner products for all candidates in  $H$  to refine the final top- $k$  results for efficient AMIPS. The second stage is *reordering*.

Algorithm 4 shows the search procedure of the approximate SINDI index. Given a query  $\vec{q}$ , inverted index  $I$ , pruning ratio  $\beta$ , reordering size  $\gamma$ , and target  $k$ , the algorithm initializes an empty min-heap  $H$  for coarse candidates and a max-heap  $R$  for the final top- $k$  results (Lines 1–2). It first derives the  $\beta$ -mass subvector  $\vec{q}'$  (Line 3) and invokes PRECISESINDISEARCH to obtain  $\gamma$  coarse candidates stored in  $H$  (Line 4). While  $H$  is not empty (Line 5), the best coarse candidate  $(i, dis)$  is popped (Line 6), its exact distance to  $\vec{q}$  is computed as  $dis'$  (Line 7), and  $(i, dis')$  is inserted into  $R$  if it improves the current set or if  $R$  contains fewer than  $k$  elements (Line 8). If  $R$  exceeds size  $k$ , its worst element is removed (Lines 9–10). After all candidates are processed,  $R$  is returned as the final result (Line 12).

## 5 EXPERIMENTAL STUDY

### 5.1 Experimental Settings

**Datasets.** Table 3 summarizes the datasets used in our experiments, covering: (i) English datasets from the MSMARCO [23] passage ranking benchmark (including SPLADE-1M and SPLADE-FULL) and the NQ [14] (Natural Questions) benchmark, all trained with the SPLADE model; (ii) Chinese dataset AntSparse, real business data from Ant Group trained with the BGE-M3 [5] model, which has higher dimensionality due to the larger Chinese vocabulary; (iii) Random datasets with non-zero entry dimensions and values drawn from a uniform distribution. For each dataset, Table 3 reports the average number of non-zero entries per vector ( $\text{avg } \|\vec{x}_i\|$ ), average vectors per inverted list ( $\text{avg } l$ ), and sparsity. The *sparsity* of  $\mathcal{D}$  is:  $\text{sparsity} = 1 - \frac{\sum_{\vec{x} \in \mathcal{D}} \|\vec{x}\|}{\|\mathcal{D}\| \cdot d}$ .

We compare SINDI with five SOTA algorithms: SEISMIC, PYANNS, SOSIA, BMP, and HNSW. Below is a description of each algorithm:

- **SEISMIC** [3]: A sparse vector index based on inverted lists.
- **SOSIA** [33]: A sparse vector index using min-hash signatures.
- **BMP** [4, 21]: A dynamic pruning strategy for learning sparse vector retrieval. It divides the original dataset into fine-grained

blocks and generates a maximum value vector for each block to evaluate whether the block should be queried.

- **HNSW** [20]: A graph-based index originally for dense vectors; we modify the data format and distance computation to support sparse vectors.
- **PYANNS** [26]: The open-source champion of the BigANN Benchmark 2023 Sparse Track. It is built on HNSW and incorporates quantization, query pruning, and rerank strategies.

**Parameter Settings.** We use the optimal parameters for each algorithm to ensure a fair comparison. Parameter choices either follow the recommendations from the original authors or are determined via grid search. Details are shown in Table 4.

**Performance Metrics.** We evaluate index construction time, index size, recall, and queries per second (QPS) for all baselines. For a query  $\vec{q}$ , let  $R = \{\vec{x}_1, \dots, \vec{x}_k\}$  denote the AMIPS results, and  $R^* = \{\vec{x}_1^*, \dots, \vec{x}_k^*\}$  denote the exact MIPS results. Recall is computed as  $\text{Recall} = \frac{\|R \cap R^*\|}{\|R^*\|}$ . We specifically report Recall@50 and Recall@100. Since approximate methods trade off efficiency and accuracy, we also report *throughput*, defined as the number of queries processed per second.

**Environment.** Experiments are conducted on a server with an Intel Xeon Platinum 8269CY CPU @ 2.50GHz and 512 GB memory. We implement SINDI in C++, compiled with g++ 10.2.1 using `-Ofast` and `AVX-512` instructions.

### 5.2 Overall Performance

**5.2.1 Recall and QPS.** Figure 8 shows the relationship between recall (Recall@50 and Recall@100) and single-threaded QPS for all algorithms. For each method, we report the best results across all tested parameter configurations.

On both English and Chinese datasets, SINDI achieves the highest QPS at the same recall levels. When Recall@50 is 99%, on SPLADE-1M, the QPS of SINDI is 2.0 $\times$  that of SEISMIC and 26.4 $\times$  that of PYANNS; on SPLADE-FULL, it is 4.16 $\times$  and 10.0 $\times$  higher, respectively. When Recall@100 is 98% on SPLADE-FULL, SINDI attains 1.9 $\times$  the QPS of SEISMIC and 3.2 $\times$  that of PYANNS.

On the Chinese AntSparse-10M dataset encoded by the BGE-M3 [5] model, SINDI also performs best. Fixing Recall@50 at 97%, its QPS is 2.5 $\times$  that of SEISMIC, the recall of PYANNS is limited due to the high *sparsity* of the dataset.

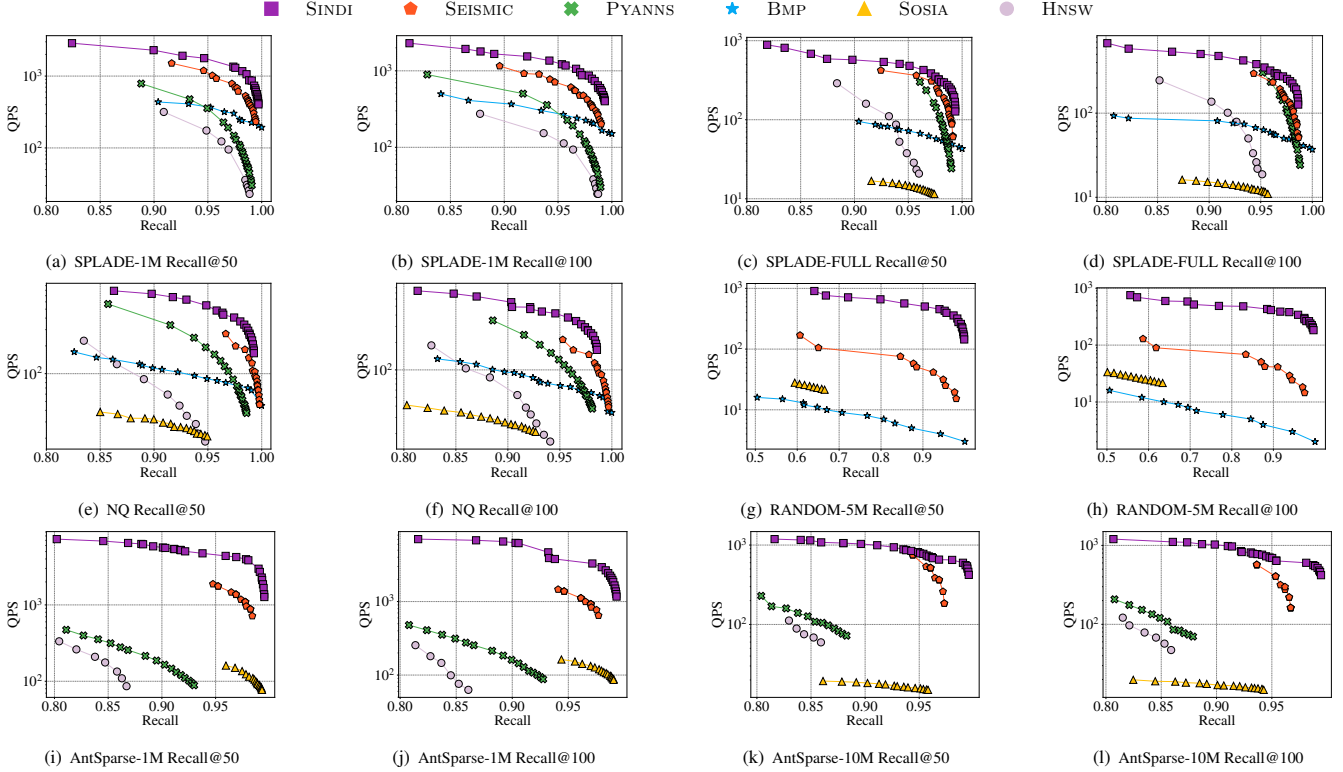
On RANDOM-5M, generated uniformly at random, the uniform distribution yields very few common non-zero dimensions between vectors, causing graph-based methods (PYANNS, HNSW) to suffer severe connectivity loss and low recall. The clustering effectiveness of SEISMIC is also sensitive to data distributions, resulting in marked degradation. In contrast, SINDI is unaffected by data distribution and achieves the best performance, with QPS exceeding SEISMIC by **an order of magnitude**.

Overall, these results demonstrate that SINDI consistently delivers state-of-the-art performance across datasets with diverse languages, models, and distributions.

**5.2.2 Index Size and Construction Time.** Figure 9 summarizes the index size and construction time of SINDI, SEISMIC, and PYANNS on the two largest datasets (SPLADE-FULL and AntSparse-10M), showing that SINDI has the lowest construction cost.

**Table 3: Dataset Statistics and Characteristics**

Dataset	$  \mathcal{D}  $	avg $  x_i  $	$nq$	avg $  q  $	$d$	Sparsity	Size (GB)	avg $l$	Model	Language
SPLADE-1M	1,000,000	126.3	6980	49.1	30108	0.9958	0.94	4569.2	SPLADE	English
SPLADE-FULL	8,841,823	126.8	6980	49.1	30108	0.9958	8.42	40447.3	SPLADE	English
AntSparse-1M	1,000,000	40.1	1000	5.8	250000	0.9998	0.31	902.6	BGE-M3	Chinese
AntSparse-10M	10,000,000	40.1	1000	5.8	250000	0.9998	3.06	6560.7	BGE-M3	Chinese
NQ	2,681,468	149.4	3452	47.0	30510	0.9951	3.01	13914.7	SPLADE	English
RANDOM-5M	5,000,000	150.0	5000	50.4	30000	0.9950	5.62	25000.0	-	-


**Figure 8: Overall Performance.**
**Table 4: Construction Parameter Settings**

Algorithm	Construction Parameter Settings
SINDI	SPLADE-FULL: $\alpha \in \{0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8\}$ AntSparse_10M: $\alpha \in \{0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1\}$
SEISMIC	SPLADE-FULL: $\lambda = 6000, \beta = 0.067, \alpha = 0.4$ AntSparse_10M: $\lambda \in \{5000, 6000, 7000, 8000, 9000, 10000, 50000\}, \beta = 0.1, \alpha = 0.5$
PYANNS	max_degree = 32, ef_construction = 1000
SOSIA	$m = 150, l = 40$
BMP	$l = 16$
HNSW	max_degree = 32, ef_construction = 1000

SEISMIC, which stores summary vectors for each block, yields the largest index size; on AntSparse-10M, its size is  $3.8\times$  that of SINDI. By contrast, the graph index construction of PYANNS requires numerous distance computations to find neighbors, resulting in a construction time  $71.5\times$  that of SINDI on SPLADE-FULL. In comparison, SINDI mainly sorts each vector’s non-zero entries for pruning, keeping the cost low and enabling rapid index building.

### 5.3 Parameters

**5.3.1 The Impact of  $\alpha$ .** This experiment examines how the document pruning parameter  $\alpha$ , which controls the proportion of high-mass non-zero entries retained, affects SINDI’s performance. A larger  $\alpha$  retains more non-zero entries per vector, potentially increasing recall but also raising the search cost. We vary  $\alpha$  from 0.4 to 0.8 (step 0.1) on SPLADE-FULL, and from 0.7 to 1.0 (step 0.05) on AntSparse, keeping  $\beta$  and  $\gamma$  fixed.

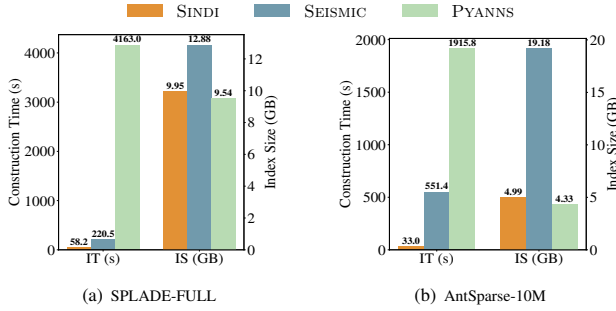


Figure 9: Index Size and Construction Time for Different Datasets and Algorithms.

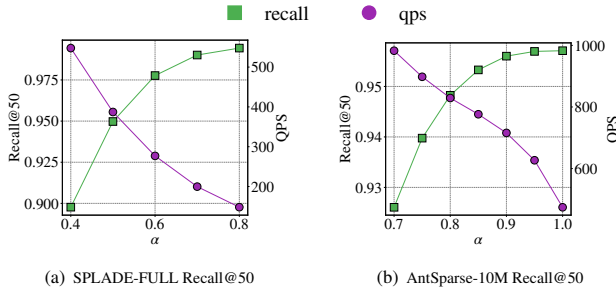


Figure 10: The Impact of  $\alpha$ .

Figure 10 shows that, on MSMARCO, recall rises and QPS drops as  $\alpha$  increases, with both trends flattening at higher  $\alpha$ . In the lower  $\alpha$  range, recall improves rapidly with moderate QPS loss, while in the higher range, recall gains slow and QPS stabilizes. On AntSparse, recall also grows more slowly at large  $\alpha$ , but QPS declines more steeply.

The slower recall gain at high  $\alpha$  is due to the *saturation effect* in Section 4.1: once enough non-zero entries are kept, further additions barely reduce inner product error. The sharper QPS drop on AntSparse comes from its lower variance of non-zero values, which leads to more retained entries for the same  $\alpha$  increase, and thus more postings to scan.

**5.3.2 The Impact of sparsity.** We evaluate the performance of SINDI under varying dataset *sparsity* levels. For a fixed  $\text{avg } \|\vec{x}\|$ , larger  $d$  yields higher *sparsity*. To examine its impact, we generate five synthetic datasets ( $|\mathcal{D}| = 1\text{M}$ ,  $\text{avg } \|\vec{x}\| = 120$ ) with  $d \in \{10\text{k}, 30\text{k}, 50\text{k}, 70\text{k}, 100\text{k}\}$ , thereby increasing *sparsity* gradually. Each dataset is generated uniformly at random, where both the positions and the values of non-zero entries follow a uniform distribution. Figure 11 compares SINDI and SEISMIC at Recall@50 = 90% and Recall@50 = 99%.

As *sparsity* increases, both SINDI and SEISMIC achieve higher QPS at the same recall because the IVF index produces shorter inverted lists (average length  $\text{avg } l$  decreases), reducing the number of candidate non-zero entries  $||q||l$  to be visited. Since all true nearest neighbors still reside in the probed lists, recall remains unaffected.

SINDI consistently outperforms SEISMIC by maintaining approximately 10 $\times$  higher QPS across all *sparsity* levels. This demonstrates SINDI’s efficiency and robustness to different data distributions.

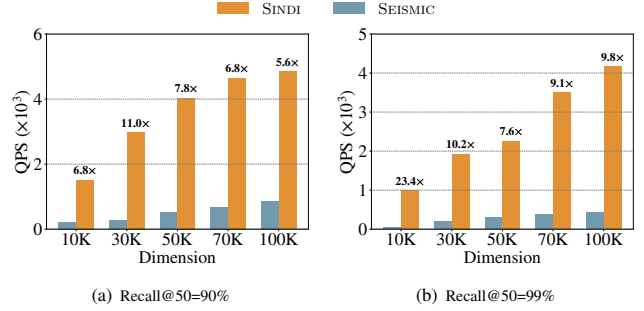


Figure 11: QPS of SINDI and SEISMIC on RANDOM-1M Dataset with Different Sparsity

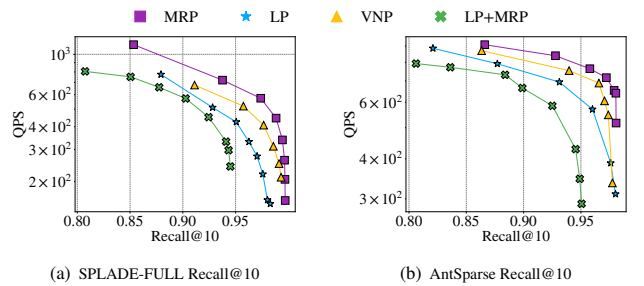


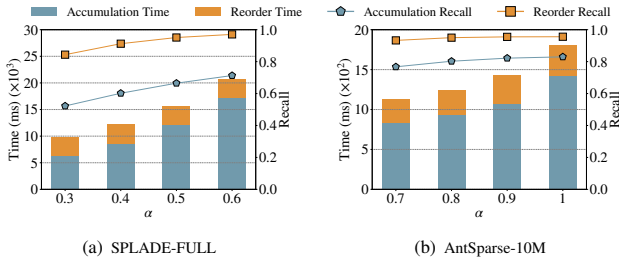
Figure 12: Recall@10 vs QPS on MSMARCO and AntSparse of Mass Ratio Pruning, List Pruning and Vector Number Pruning.

## 5.4 Ablation

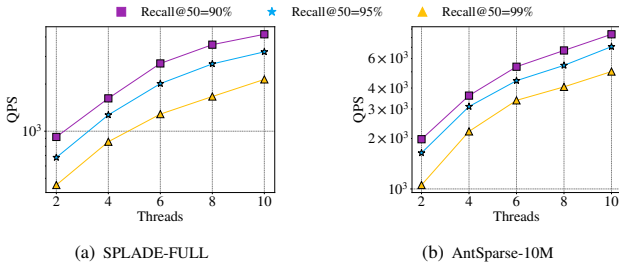
**5.4.1 The Impact of Pruning Method.** Figure 12 illustrates the performance of different pruning strategies on the SPLADE-FULL and AntSparse datasets. All strategies are evaluated under the same  $\beta$  and  $\gamma$  settings, while varying  $\alpha$  to measure Recall and QPS. *Mass Ratio Pruning* achieves the best overall performance, followed by *Vector Number Pruning* and *List Pruning*, with the lowest performance observed when combining *List Pruning* and *Mass Ratio Pruning*.

The advantage of *Mass Ratio Pruning* lies in its ability to preserve the non-zero entries that contribute most to the inner product, thereby retaining more true nearest neighbors during the coarse recall stage. In contrast, *List Pruning* limits the posting list size for each dimension, which can result in two issues: some lists become too short and keep mainly small-value entries, while others remain too long and remove large-value entries. As a result, *List Pruning* is less suitable for SINDI. SEISMIC, however, uses *List Pruning* because it computes the full inner product for all vectors in the lists, avoiding large accuracy losses. When *List Pruning* is combined with *Mass Ratio Pruning*, even more non-zero entries are discarded, further reducing recall.

**5.4.2 The Impact of Reorder.** To investigate the impact of the reordering strategy on the performance of SINDI, we compared the cases with and without reordering on the SPLADE-FULL and AntSparse datasets. The query prune ratio  $\beta$  was set to 0.2, and the reordering number  $\gamma$  was fixed at 500. For the document prune ratio  $\alpha$ , we varied it from 0.3 to 0.6 in steps of 0.1 on SPLADE-FULL,



**Figure 13: Reorder vs. Non-Reorder on SPLADE-FULL and AntSparse-10M Datasets: Time Cost and Recall@50 with Varying  $\alpha$ .**



**Figure 14: Multi-threaded Scalability of SINDI (QPS) at Different Recall@50 Targets on SPLADE-FULL and AntSparse-10M Datasets.**

and from 0.7 to 1.0 in steps of 0.1 on AntSparse-10M. The non-reordering strategy does not require storing the dataset. We evaluated both strategies using query time and Recall@50 as performance metrics.

The results are shown in Figure 13. For the reordering strategy, the query time includes both accumulation time and reordering time, whereas the non-reordering strategy includes only accumulation time. Since  $\gamma$  is fixed, the reorder time remains relatively constant. As  $\alpha$  increases, the accumulation time also increases accordingly. Although reorder time accounts for only a small portion of the total query time, it yields a substantial recall improvement. For example, on SPLADE-FULL with  $\alpha = 0.6$ , the accumulation time is 17099 ms and the reorder time is 3553 ms ( $\approx 17.2\%$  of the total), yet recall improves from 0.71 to 0.97. This demonstrates the clear benefit of incorporating the reordering strategy.

The reordering strategy is effective for two main reasons. First, it focuses computation on a limited set of non-zero entries that contribute most to the inner product, greatly reducing unnecessary operations. Second, the partial inner products derived from high-value entries largely preserve the true ranking order, ensuring that small  $\gamma$  is sufficient to contain the true nearest neighbors, thereby improving both efficiency and accuracy.

## 5.5 Scalability

To further evaluate the scalability of the SINDI algorithm, we conducted a multi-threaded performance test on two large-scale datasets: SPLADE-FULL and AntSparse-10M. We measured QPS at different recall targets, with Recall@50  $\in \{0.95, 0.97, 0.99\}$  for SPLADE-FULL and Recall@50  $\in \{0.90, 0.95, 0.99\}$  for AntSparse-10M, while varying the number of CPU cores from 2 to 10, as shown in Figure 14.

On AntSparse-10M at Recall@50 = 0.90, using 2 CPU cores yields 1979.49 QPS ( $\approx 989.75$  QPS per core), whereas 10 cores achieve 8374.01 QPS ( $\approx 837.40$  QPS per core), indicating per-core efficiency drops by less than 16% when scaling from 2 to 10 cores. On SPLADE-FULL at Recall@50 = 0.99, using 2 CPU cores yields 453.46 QPS ( $\approx 226.73$  QPS per core), whereas 10 cores achieve 2142.05 QPS ( $\approx 214.21$  QPS per core), corresponding to a per-core efficiency drop of about 5.5%. Similar scaling behavior is observed across other recall targets for both datasets.

These results show that SINDI maintains high multi-core efficiency across datasets and accuracy targets, confirming its suitability for deployment in scenarios requiring both high recall and high throughput, with minimal parallelization overhead.

## 6 RELATED WORK

Existing methods for MIPS on sparse vectors employ various index structures, including inverted index, graph, and hash-based designs. Inverted index-based methods, such as SEISMIC [3], maintain postings for each dimension and prune low-value entries, often clustering postings into blocks, but each vector may appear in multiple postings, causing random access to scattered memory and reducing locality.

Graph-based methods, such as the HNSW-based PYANNS [20], organize vectors as nodes in a proximity graph, but vector sparsity weakens connectivity and greedy traversal still incurs frequent random memory accesses.

Hash-based methods, like SOSIA [33], transform sparse vectors into sets via the SOS representation and apply min-hash to estimate Jaccard similarity. Multiple hash functions improve accuracy but scatter storage across many buckets, hurting locality and making direct computation inefficient.

BMP [4, 21] stores non-zero values in postings and prunes them using block-level maximum value vectors. While this avoids common-dimension lookups, overly fine-grained block partitioning increases the number of candidate blocks to evaluate, lengthening query time. In contrast, our method keeps value-stored postings without incurring such overhead.

## 7 CONCLUSION

In this work, we propose SINDI, an inverted index for sparse vectors that eliminates redundant inner-product computations. By storing non-zero entries directly in the postings, SINDI removes both document ID lookups and random memory accesses, and leverages SIMD acceleration to maximize CPU parallelism. It further introduces *Mass Ratio Pruning*, which effectively preserves high-value entries, and a reordering strategy whose refinement step ensures high accuracy. Experiments on multilingual, multi-scale real-world datasets demonstrate that SINDI delivers state-of-the-art performance in both recall and throughput.

## REFERENCES

- [1] Firas Abuzaid, Geet Sethi, Peter Bailis, and Matei Zaharia. 2019. To index or not to index: Optimizing exact maximum inner product search. In *Proceedings of the 2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1250–1261.
- [2] Dimitri Bertsekas and Robert Gallager. 2021. *Data Networks*. Athena Scientific.
- [3] 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*. 152–162.

- [4] Parker Carlson, Wentai Xie, Shanxiu He, and Tao Yang. 2025. Dynamic superbloc pruning for fast learned sparse retrieval. In *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval* (Padua, Italy) (SIGIR '25). Association for Computing Machinery, New York, NY, USA, 3004–3009. <https://doi.org/10.1145/3726302.3730183>
- [5] Jianyu Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. 2024. M3-embedding: Multi-linguality, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. In *Findings of the Association for Computational Linguistics: ACL 2024* (Bangkok, Thailand). Association for Computational Linguistics, 2318–2335. <https://doi.org/10.18653/v1/2024.findings-acl.137>
- [6] N. R. Draper. 1998. *Applied regression analysis*. McGraw-Hill Inc.
- [7] 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* (Madrid, Spain) (SIGIR '22). Association for Computing Machinery, New York, NY, USA, 2353–2359. <https://doi.org/10.1145/3477495.3531857>
- [8] Thibault Formal, Carlos Lassance, Benjamin Piwowarski, and Stéphane Clinchant. 2024. Towards effective and efficient sparse neural information retrieval. *ACM Transactions on Information Systems* 42, 5, Article 116 (April 2024). <https://doi.org/10.1145/3634912>
- [9] Thibault Formal, Benjamin Piwowarski, and Stéphane Clinchant. 2021. SPLADE: Sparse lexical and expansion model for first stage ranking. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval* (Virtual Event, Canada) (SIGIR '21). Association for Computing Machinery, New York, NY, USA, 2288–2292. <https://doi.org/10.1145/3404835.3463098>
- [10] John L. Hennessy and David A. Patterson. 2011. *Computer architecture: A quantitative approach*. Elsevier.
- [11] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, 604–613.
- [12] Intel Corporation. 2025. Intel VTune Profiler. <https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html> Online; accessed 10 June 2025.
- [13] Omid Keivani, Kaushik Sinha, and Parikshit Ram. 2018. Improved maximum inner product search with better theoretical guarantee using randomized partition trees. *Machine Learning* 107, 6 (June 2018), 1069–1094. <https://doi.org/10.1007/s10994-018-5711-7>
- [14] 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. [https://doi.org/10.1162/tac1\\_a\\_00276](https://doi.org/10.1162/tac1_a_00276)
- [15] 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). Association for Computing Machinery, New York, NY, USA, 2220–2226. <https://doi.org/10.1145/3477495.3531833>
- [16] Charles L. Lawson and Richard J. Hanson. 1995. *Solving least squares problems*. SIAM.
- [17] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, Vol. 33. Curran Associates, Inc., 9459–9474. [https://proceedings.neurips.cc/paper\\_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf](https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf)
- [18] Linux Kernel Organization. 2025. perf: Linux Profiling with Performance Counters. [https://perf.wiki.kernel.org/index.php/Main\\_Page](https://perf.wiki.kernel.org/index.php/Main_Page) Online; accessed 10 June 2025.
- [19] Guangyuan Ma, Yongliang Ma, Xuanrui Gou, Zhenpeng Su, Ming Zhou, and Songlin Hu. 2025. LightRetriever: A LLM-based hybrid retrieval architecture with 1000x faster query inference. arXiv:2505.12260. <https://arxiv.org/abs/2505.12260>
- [20] Yu A. Malkov and D. 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 (April 2020), 824–836. <https://doi.org/10.1109/TPAMI.2018.2889473>
- [21] Antonio Mallia, Torsten 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* (Washington DC, USA) (SIGIR '24). Association for Computing Machinery, New York, NY, USA, 2411–2415. <https://doi.org/10.1145/3626772.3657906>
- [22] Behnam Neyshabur and Nathan Srebro. 2015. On symmetric and asymmetric LSHs for inner product search. In *Proceedings of the 32nd International Conference on Machine Learning - Volume 37 (ICML '15)*. JMLR.org, 1926–1934.
- [23] Tri Nguyen, Mir Rosenberg, Xia Song, Jianfeng Gao, Saurabh Tiwary, Rangan Majumder, and Li Deng. 2016. MS MARCO: A human generated machine reading comprehension dataset. In *Proceedings of the Workshop on Cognitive Computation: Integrating Neural and Symbolic Approaches 2016 co-located with the 30th Annual Conference on Neural Information Processing Systems (NIPS 2016) (CEUR Workshop Proceedings)*, Vol. 1773. CEUR-WS.org. [https://ceur-ws.org/Vol-1773/CoCoNIPS\\_2016\\_paper9.pdf](https://ceur-ws.org/Vol-1773/CoCoNIPS_2016_paper9.pdf)
- [24] Ninh Pham. 2021. Simple yet efficient algorithms for maximum inner product search via extreme order statistics. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining* (Virtual Event, Singapore) (KDD '21). Association for Computing Machinery, New York, NY, USA, 1339–1347. <https://doi.org/10.1145/3447548.3467345>
- [25] Idan Pogrebinsky, David Carmel, and Oren Kurland. 2025. Enhancing retrieval-augmented generation for text completion through query selection. In *Proceedings of the ACM International Conference on the Theory of Information Retrieval* (Padua, Italy) (ICTIR '25). Association for Computing Machinery, New York, NY, USA, 410–415. <https://doi.org/10.1145/3731120.3744610>
- [26] PyANNS Contributors. 2025. PyANNS: C++ code for approximate nearest neighbor search. <https://github.com/veaanaab/pyanns> GitHub repository.
- [27] Derrick Quinn, Mohammad Nouri, Neel Patel, John Salihu, Alireza Salemi, Sukhan Lee, Hamed Zamani, and Mohammad Alian. 2025. Accelerating retrieval-augmented generation. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) (ASPLOS '25). Association for Computing Machinery, New York, NY, USA, 15–32. <https://doi.org/10.1145/3669940.3707264>
- [28] Tolga Şakar and Hakan Emekci. 2025. Maximizing rag efficiency: A comparative analysis of rag methods. *Natural Language Processing* 31, 1 (2025), 1–25.
- [29] Kunal Sawarkar, Abhilasha Mangal, and Shivam Raj Solanki. 2024. Blended rag: Improving rag (retriever-augmented generation) accuracy with semantic search and hybrid query-based retrievers. In *Proceedings of the 2024 IEEE 7th International Conference on Multimedia Information Processing and Retrieval (MIPR)*. IEEE, 155–161.
- [30] Yang Song, Yu Gu, Rui Zhang, and Ge Yu. 2021. Promips: Efficient high-dimensional c-approximate maximum inner product search with a lightweight index. In *Proceedings of the 2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1619–1630.
- [31] Weihang Su, Yichen Tang, Qingyao Ai, Junxi Yan, Changyue Wang, Hongning Wang, Ziyi Ye, Yujia Zhou, and Yiqun Liu. 2025. Parametric retrieval augmented generation. In *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval* (Padua, Italy) (SIGIR '25). Association for Computing Machinery, New York, NY, USA, 1240–1250. <https://doi.org/10.1145/3726302.3729957>
- [32] Xiao Yan, Jinfeng Li, Xinyan Dai, Hongzhi Chen, and James Cheng. 2018. Norm-ranging LSH for maximum inner product search. In *Advances in Neural Information Processing Systems*, Vol. 31. Curran Associates, Inc. [https://proceedings.neurips.cc/paper\\_files/paper/2018/file/b60c5ab647a27045b462934977ccad9a-Paper.pdf](https://proceedings.neurips.cc/paper_files/paper/2018/file/b60c5ab647a27045b462934977ccad9a-Paper.pdf)
- [33] Xi Zhao, Zhonghan Chen, Kai Huang, Ruiyuan Zhang, Bolong Zheng, and Xiaofang Zhou. 2024. Efficient approximate maximum inner product search over sparse vectors. In *Proceedings of the 2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 3961–3974.
- [34] Xi Zhao, Bolong Zheng, Xiaomeng Yi, Xiaofan Luan, Charles Xie, Xiaofang Zhou, and Christian S. Jensen. 2023. FARGO: Fast maximum inner product search via global multi-probing. *Proceedings of the VLDB Endowment* 16, 5 (Jan. 2023), 1100–1112. <https://doi.org/10.14778/3579075.3579084>