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

Ruoxuan Li [1], Xiaoyao Zhong [2], Jiabao Jin [2], Peng Cheng [1,3], Wangze Ni [4], Zhitao Shen [2], Wei Jia [2], Xiangyu Wang [2], Heng Tao Shen [3], Jingkuan Song [3]

[1]*East China Normal University, Shanghai, China;* [2]*Ant Group, Shanghai, China;*
[3]*Tongji University, Shanghai, China;* [4]*Zhejiang University, Hangzhou China*

rxlee@stu.ecnu.edu.cn, zhongxiaoyao.zxy@antgroup.com, jinjiabao.jjb@antgroup.com,
cspcheng@tongji.edu.cn, niwangze@zju.edu.cn, zhitao.szt@antgroup.com, jw94525@antgroup.com,
wxy407827@antgroup.com, 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 $4.2\times$ to $26.4\times$ compared with Seismic and PyANNs. Notably, SINDI has been integrated into Ant Group's open-source vector search library, *VSAG*.

***Index Terms*—Maximum Inner Product, Sparse Vectors**

I. Introduction

Recently, retrieval-augmented generation (RAG) [1]–[6] has become one of the 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 have found that complementing dense vector-based RAG with sparse vector retrieval yields superior accuracy and recall [7]–[10]. Unlike dense vectors, sparse vectors (where only a small fraction of dimensions are non-zero) are generated by specific models (e.g., SPLADE [11]–[14]) to preserve semantic information while enabling precise lexical matching [7]. In the enhanced RAG framework, dense vectors capture holistic semantic similarity, while sparse vectors ensure exact term recall. This synergy translates into significant application-level improvements: in the evaluation of AntGroup production RAG system, integrating sparse vectors into a strong hybrid baseline (BM25 + Dense) boosted **Recall@3 by 18.5%** (from 55.60% to 74.10%) and **Recall@10 by 3.7%** (from 85.20% to 88.90%). We illustrate the process of this enhanced RAG workflow in the following example:

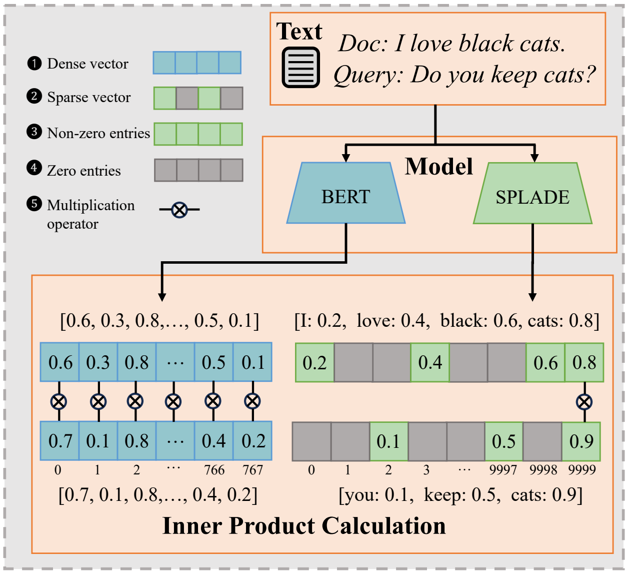


Fig. 1: Example of Dense and Sparse Vector Representations and Inner Product Calculations.

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 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) [15]–[19], 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 [20], performing exact MIPS in high-dimensional spaces is computationally prohibitive. To mitigate this issue, we focus on Approximate Maximum Inner Product Search (AMIPS) [17], [21], [22], which trades a small amount of recall for significantly improved search efficiency.

Many algorithms [23]–[26] have been proposed for AMIPS, employing techniques such as inverted index [23], proximity graphs [26], and hash-based [24] 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.

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 I. Regarding the computational cost, Sindi achieves an **amortized time complexity** of $O\left(\frac{\|q\|}{s}\right)$ for computing the inner product between a query q and a document x in **full-precision scenarios** (where symbols are defined in Table II). In contrast, traditional inverted index and graph-based algorithms typically scale with $O(\|q\| + \|x\|)$. The detailed derivation is given in § III-B.

TABLE I: 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

TABLE II: 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)
λ	window size
σ	number of windows
α	base vector pruning ratio
β	query vector pruning ratio
γ	reorder pool size
$I,\ I_j$	inverted index; inverted list for dimension j
$I_{j,w}$	w-th window of inverted list I_j
$P^j, P^j[t]$	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 $\vec{x}_2$

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

- We present Sindi, a novel value-storing inverted index described in §III-A and §III-B, which reduces redundant distance computation and random memory accesses. We further introduce a *Window Switch* strategy in §III-C to support large-scale datasets.
- We propose *Vector Mass Pruning* in §IV to decrease the search space and improve query speed while maintaining accuracy.
- We evaluate Sindi on multi-scale, multilingual datasets in §V, demonstrating 4× ∼ 26× higher single-thread Queries Per Second (QPS) than PyANNS and Seismic at over 99% Recall@50, and achieving 8.8M-scale index construction in 60 seconds with minimal cost.

II. Preliminaries

A. 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:* $x = \{\, x^j \mid x^j \neq 0,\ j \in [0, d-1] \,\}$*. 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: $[I : 0.2, love : 0.4, black : 0.6, cats : 0.8]$. 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., $\vec{x}^* = \underset{\vec{x} \in \mathcal{D}}{\arg\max}\ \delta(\vec{x}, \vec{q})$.*

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.

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 II, which will be referred to throughout the rest of the paper.

B. Existing Solutions

Representative algorithms for the AMIPS problem on sparse vectors include the inverted-index based SEISMIC [23], 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.

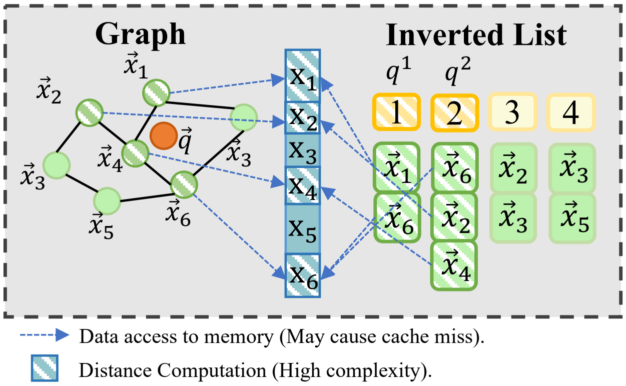


Fig. 2: The Bottleneck of the Graph Index and Inverted Index During Searching Process.

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 [27] and VTune [28] 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 organize 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%.

III. FULL PRECISION INVERTED INDEX

This section introduces full-precision SINDI, an inverted index designed for sparse vector retrieval. Its advantages

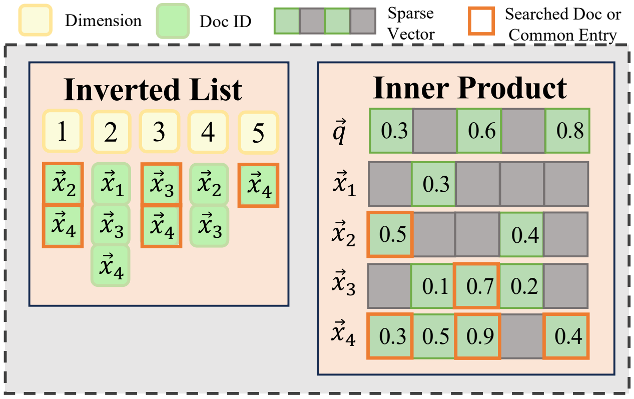


Fig. 3: Overlap of Inverted List Entries and Common Non-Zero Dimensions in Inner-Product Computation.

are organized along three aspects: index structure, distance computation, and cache optimization.

- In § III-A, 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 § III-B, 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 yielding superior CPU utilization compared to state-of-the-art methods and improves query throughput.
- In § III-C, to reduce cache misses, SINDI implements a Window Switch strategy. It partitions posting lists into fixed-size segments (λ) that share a distance array. This approach minimizes memory overhead, and as shown theoretically and in Figure 5, an optimal λ exists to minimize memory access costs.

A. 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 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.

B. 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 P^j. Here, P^j holds the products for I_j, with each entry $P^j[t]$ corresponding to the t-th entry in I_j. Therefore, P^j has the same length as I_j.

In the second stage, the values in P^j are aggregated into a preallocated 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 $size(A) = 9$ 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 $P[0]$. Accumulating $P[0]$ 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 $P[2]$, and add it to obtain $A[4] = 31.0$. The computation for I_8 proceeds analogously. Finally, $A[4]$ becomes 36.1, which 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 P^j. This approach maximizes CPU utilization and reduces the complexity of the

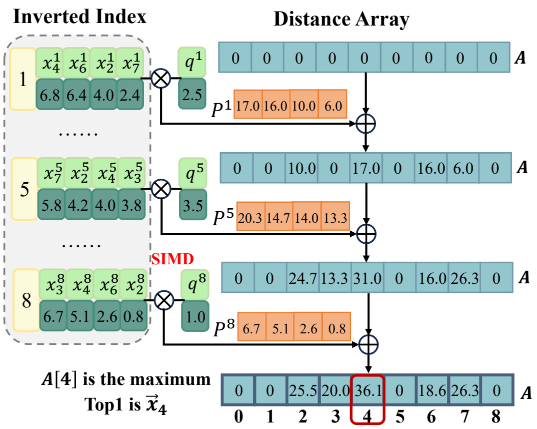


Fig. 4: An Example of SINDI and Query Process.

inner product computation from $O(\|q\| + \|x\|)$ to $O\left(\frac{\|q\|}{s}\right)$, where s denotes the number of elements processed per SIMD operation. The complexity of SINDI's distance computation is derived as follows:

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

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

C. 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.

Algorithm 1: PRECISESINDICONSTRUCTION

```
Input: A sparse dataset $\mathcal{D}$ and dimension $d$, window
       size $\lambda$
Output: Inverted list $I$
for $j \in \{0, ..., d-1\}$ do
    $\mathcal{X} \leftarrow \{\, \vec{x}_i \in \mathcal{D} \mid x_i^j \neq 0 \,\}$;
    foreach $\vec{x}_i \in \mathcal{X}$ do
        $w \leftarrow \lfloor \frac{i}{\lambda} \rfloor$
        $I_{j,w}.append(x_i^j)$
return $I$
```

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 λ $(0 < \lambda \leq \|\mathcal{D}\|)$, and the total number of windows is $\sigma = \left\lceil \frac{\|\mathcal{D}\|}{\lambda} \right\rceil$. 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 $\left\lfloor \frac{i}{\lambda} \right\rfloor$. Each inverted list I_j is partitioned in the same way, so every list has σ 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 λ, 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 P^j, which has the same length as $I_{j,w}$ and is index-aligned with it — i.e., $P^j[t]$ stores the product for the t-th posting in $I_{j,w}$. The accumulation stage then adds each $P^j[t]$ 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 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 $O\left(\frac{\|q\|}{s}\right)$.

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 $O(\|\mathcal{D}\|\|\bar{x}\|)$, where $\|\bar{x}\| = \frac{\sum_{\vec{x}_i \in \mathcal{D}} \|x_i\|}{\|\mathcal{D}\|}$ 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 λ is initialized to zeros (Lines 1–2) and an empty min-heap H is created (Line 3). The outer loop traverses all windows $w \in \{0, \ldots, \sigma - 1\}$ (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 P^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 $P^j[t]$ 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 λ.

D. 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 λ:

- **Random-access cost to the distance array.** During accumulation, each partial score $P^j[t]$ is added to $A[i \bmod \lambda]$, producing a random write pattern over A. When λ 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 λ 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.

```
Algorithm 2: PRECISESINDISEARCH
   Input: Query $\vec{q}$, an inverted list $I$, and $k$
   Output: At most $k$ points in $\mathcal{D}$
1  for $m \in \{0, ..., \lambda - 1\}$ do
2      $A[m] \leftarrow 0$
3  $H \leftarrow$ empty min-heap
4  for $w \in \{0, ..., \sigma - 1\}$ do
5      foreach $q^j \in q$ do
6          $P^j \leftarrow$ SIMDProduct($q^j$, $I_{j,w}$);
7          for $t \in \{0, \ldots, I_{j,w}.size() - 1\}$ do
8              $x_i^j \leftarrow I_{j,w}[t]$;
9              $m \leftarrow i \bmod \lambda$;
10             $A[m] \leftarrow A[m] + P^j[t]$;
11     for $m \in \{0, ..., \lambda - 1\}$ 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 $H$
```

Selecting λ 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:

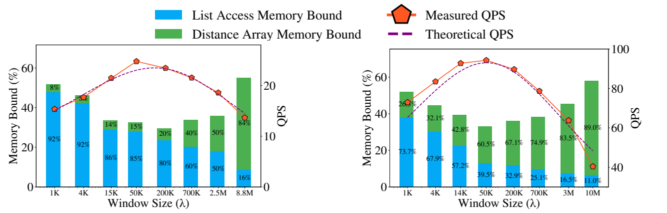
Example 6. *Figure 5 reports experimental results for the full-precision SINDI on the SPLADE-FULL and AntSparse-10M datasets. For each dataset, we executed queries under different window sizes λ and measured the QPS. We also used the Intel VTune Profiler [28] to record memory bound metrics for two types 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-FULL dataset, as λ increases from $1K$ to $8.8M$, the memory bound from distance array accesses increases monotonically, whereas that from sub-list cache evictions decreases monotonically. The total memory-bound latency reaches its minimum near $\lambda \approx 150K$, corresponding to the highest query throughput. The AntSparse-10M 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 [29], [30]:

$$T_{\text{mem}}(\lambda) = c_{rand}\lambda^{+\alpha} + c_{evict}\lambda^{-\beta} + C_0, \tag{1}$$

where:

- λ is the window size;



(a) SPLADE-FULL

(b) AntSparse-10M

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

- $c_{rand}\lambda^{+\alpha}$ models the increasing cost of random accesses to the distance array as λ grows;
- $c_{evict}\lambda^{-\beta}$ models the decreasing cost of cache eviction to sub-lists with larger λ;
- C_0 is the baseline memory-access cost unrelated to λ.

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

Example 7. *Figure 5 presents the theoretical QPS curves, where parameters (α, β) are estimated via log–log regression [31] and $(c_{rand}, c_{evict}, C_0)$ via least-squares fitting [32] of the double power-law model. For SPLADE-FULL, the model predicts an optimal $\lambda^* \approx 1.2\times10^5$, while for AntSparse-10M, it predicts $\lambda^* \approx 5.1\times10^4$. Both predictions align with the robust high-throughput interval $\lambda \in [50,000, 100,000]$. This broad plateau confirms that within this range, performance is stable, rendering precise point-wise tuning unnecessary despite minor hardware-level variability.*

IV. 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.

A. 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 [13]. 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 $\vec{x}_i'$ denote the pruned version of document $\vec{x}_i$, and $\vec{q}'$ the pruned version of query $\vec{q}$. 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 $\|q\| \cdot l \; - \; \|q'\| \cdot l'$,

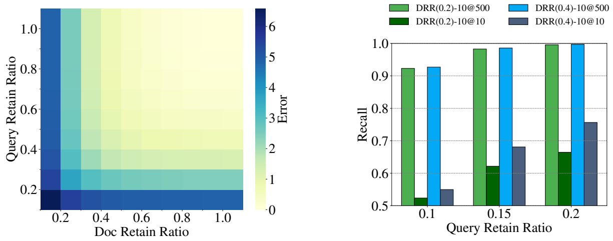
For a given pruning operator ϕ, we define the *inner product error* for document $\vec{x}_i$ as $e_i^{(\phi)} = \delta(\vec{x}_i, \vec{q}) - \delta\big(\phi(\vec{x}_i), \phi(\vec{q})\big)$, where $\delta(\cdot,\cdot)$ denotes the exact inner product and $\phi(\cdot)$ applies the pruning transformation. The total inner product error over the dataset $\mathcal{D}$ is then $\varepsilon^{(\phi)} = \sum_{\vec{x}_i \in \mathcal{D}} e_i^{(\phi)}$

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 III, for full-precision SINDI the upper bound for computing the inner product between $\vec{x}_i$ and $\vec{q}$ is $\Theta\left(\frac{\|q\|}{s}\right)$. For a given $\vec{q}$, the overall query complexity is $O\left(\frac{\|q\|\, l}{s}\right)$, 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.

As discussed in Section III, for full-precision SINDI the upper bound for computing the inner product between $\vec{x}_i$ and $\vec{q}$ is $\Theta\left(\frac{\|q\|}{s}\right)$. For a given $\vec{q}$, the overall query complexity is $O\left(\frac{\|q\|\, l}{s}\right)$, 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 retaining entries with the largest magnitude. While magnitude-based pruning is an established concept—exemplified by *List Pruning (LP)* in SEISMIC [23] and utilized in BMP [25]—SINDI's *Mass Ratio Pruning (MRP)* introduces a distinct objective. Unlike prior methods that employ pruning primarily to shrink the search space (filtering), MRP is architected to minimize **inner product error** via an adaptive cumulative mass ratio, ensuring high-quality candidates for reordering. In the following, we



(a) Inner Product Error (b) Recall Comparision

Fig. 6: Intuition of Pruning and Reorder

systematically compare LP, *Vector Number Pruning (VNP)*, and MRP to analyze their respective trade-offs.

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

Vector Number Pruning (VNP). VNP applies the pruning operator ϕ_{VNP} at the vector level. For each document vector $\vec{x}_i$, $\phi_{\text{VNP}}(\vec{x}_i)$ retains the vn non–zero entries with the largest absolute values, ensuring $\|\phi_{\text{VNP}}(\vec{x}_i)\| = vn$. 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 ϕ_{MRP} based on the cumulative sum of the absolute values of a vector's non-zero entries. For each document vector $\vec{x}_i$, $\phi_{\text{MRP}}(\vec{x}_i)$ ranks all non-zero entries in descending order of absolute value and retains the shortest prefix whose cumulative sum reaches a fraction α 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: $mass(\vec{x}) = \sum_{x^j \in x} |x^j|$.

We define the vector obtained after Mass Ratio Pruning as the α-mass subvector.

Definition 6 (α-Mass Subvector)**.** Let $\vec{x} \in \mathbb{R}^d$ and let π 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 $1 \leq r \leq \|x\|$ be the smallest integer satisfying $\sum_{j=1}^{r} |x^{\pi_j}| \geq \alpha \times mass(\vec{x})$. The α-mass subvector [23], denoted α-$mass(\vec{x})$, is the vector whose non-zero entries are $\{x^{\pi_j}\}_{j=1}^{r}$.

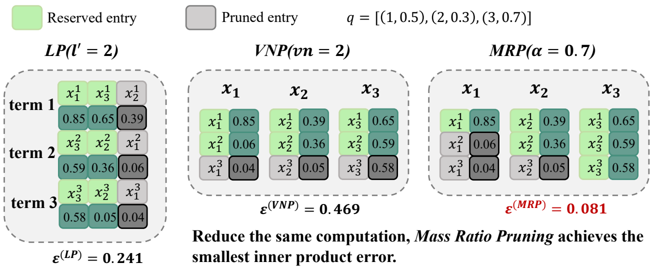


Fig. 7: An Example of *List Pruning*, *Vector Number Pruning* and *Mass Ratio Pruning*.

```
Algorithm 3: ApproximateSindiConstruction
   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      $\vec{x}'_i \leftarrow \alpha$-$mass(\vec{x}_i)$
4      $\mathcal{D}' \leftarrow \mathcal{D}' \cup \vec{x}'_i$
5  $I = $ PreciseSindiConstruction$(\mathcal{D}', d, \lambda)$
6  return $I$ and $\mathcal{D}$
```

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 α-$mass(\vec{x}_i)$ with $\alpha = 0.7$. The figure shows: (i) all three strategies result in the same reduction in computation, $\|q\|l - \|q'\|l' = 9 - 6 = 3$; (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 λ, and pruning ratio α, 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 α-mass subvector α-$mass(\vec{x}_i)$ is computed and assigned to $\vec{x}'_i$ (Line 3), which is then added to $\mathcal{D}'$. The remaining steps invoke PreciseSindiConstruction (Algorithm 1) 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 β-$mass(\vec{q})$ and is used in coarse retrieval.

Algorithm 4: APPROXIMATESINDISEARCH

```
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  $H \leftarrow$ 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 $R$
```

B. 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 pruned index to retrieve γ 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 β, reordering size γ, 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 β-mass subvector $\vec{q}'$ (Line 3) and invokes PRECISESINDISEARCH (Algorithm 2) to obtain γ 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).

V. EXPERIMENTAL STUDY

A. Experimental Settings

Datasets. Table III summarizes the datasets used in our experiments, covering: (i) English datasets from the MSMARCO [33] passage ranking benchmark (including SPLADE-1M and SPLADE-FULL) and the NQ [34] (Natural Questions) benchmark, all trained with the SPLADE model; (ii) Chinese dataset AntSparse, real business data from Ant Group trained with the BGE-M3 [35] 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 III reports the average number of non-zero entries per vector (avg $\|\vec{x}_i\|$), average vectors per inverted list (avg l), and sparsity. The *sparsity* of $\mathcal{D}$ is: $sparsity = 1 - \frac{\sum_{\vec{x}\in\mathcal{D}} \|\vec{x}\|}{\|\mathcal{D}\|\cdot d}$.

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

- **SEISMIC** [23]: A sparse vector index based on inverted lists.
- **SOSIA** [24]: A sparse vector index using min-hash signatures.
- **BMP** [25], [36]: 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** [37]: 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.
- **SHNSW** [38]:The open-source runner-up of the BigANN Benchmark 2023 Sparse Track. It is a graph-based index (HNSW variant) that utilizes memory optimization and early termination.
- **SINNAMON** [39]: An inverted index using hashing to compress vectors into dense sketches for SIMD scoring.

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.

Performance Metrics. We evaluate index construction time, index size, recall, and queries per second (QPS) for all baselines. For a query $\vec{q}$, let $R = \{\vec{x}_1, \ldots, \vec{x}_k\}$ denote the AMIPS results, and $R^* = \{\vec{x}^*_1, \ldots, \vec{x}^*_k\}$ denote the exact MIPS results. Recall is computed as 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.

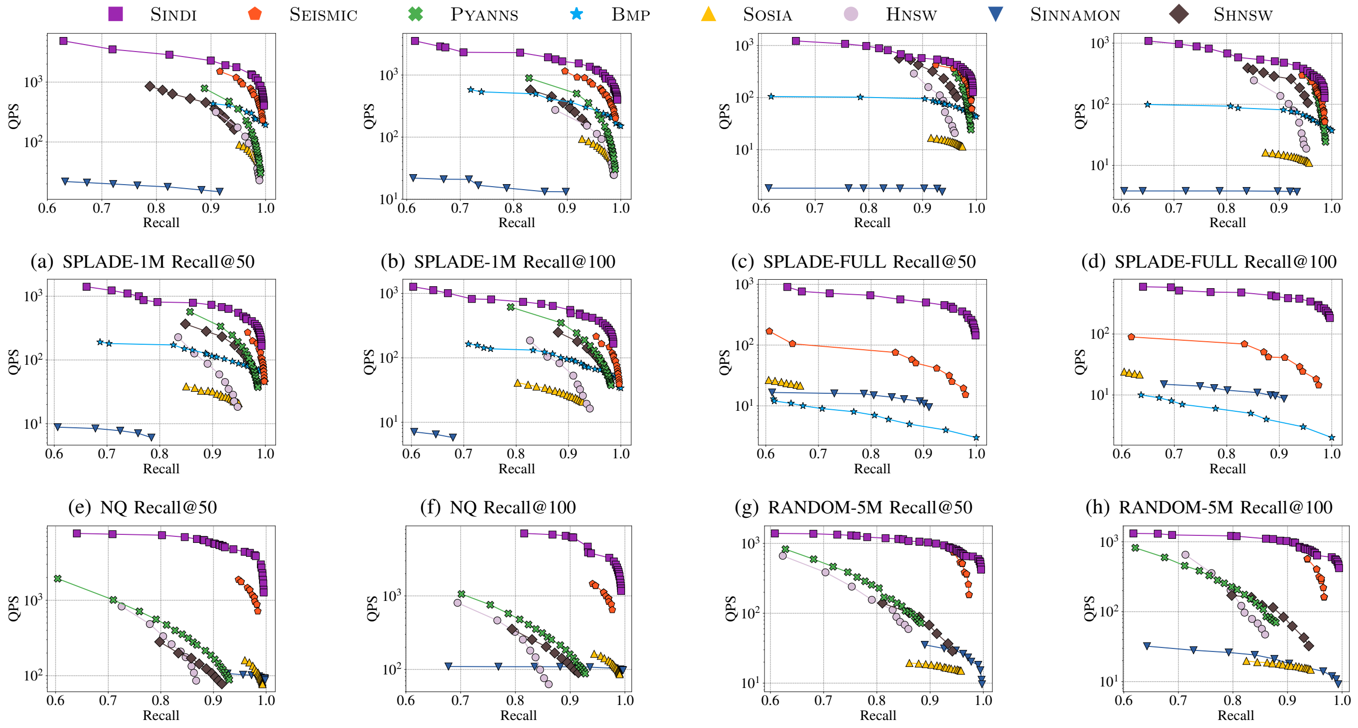
Environment. Experiments are conducted on a server running the Cent operating system, with an Intel Xeon Platinum 8269CY CPU @ 2.50GHz and 512 GB memory. We implement SINDI in C++, compiled with `g++` 10.2.1 using `-Ofast` and `AVX-512` instructions.

B. Overall Performance

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.

TABLE III: 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	-	-



(a) SPLADE-1M Recall@50

(b) SPLADE-1M Recall@100

(c) SPLADE-FULL Recall@50

(d) SPLADE-FULL Recall@100

(e) NQ Recall@50

(f) NQ Recall@100

(g) RANDOM-5M Recall@50

(h) RANDOM-5M Recall@100

(i) AntSparse-1M Recall@50

(j) AntSparse-1M Recall@100

(k) AntSparse-10M Recall@50

(l) AntSparse-10M Recall@100

Fig. 8: Overall Performance.

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× that of SEISMIC and 26.4× that of PYANNS; on SPLADE-FULL, it is 4.16× and 10.0× higher, respectively. When Recall@100 is 98% on SPLADE-FULL, SINDI attains 1.9× the QPS of SEISMIC and 3.2× that of PYANNS.

On the Chinese AntSparse-10M dataset encoded by the BGE-M3 [35] model, SINDI also performs best. Fixing Recall@50 at 97%, its QPS is 2.5× that of SEISMIC, the recall of PYANNS is limited due to the high *sparsity* of the dataset.

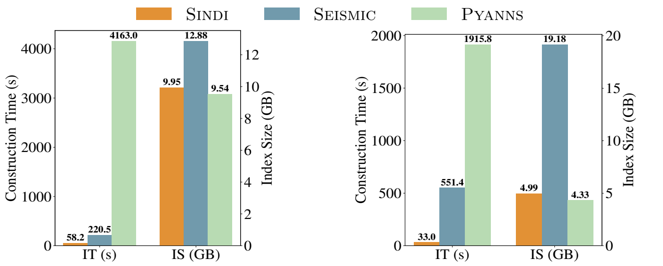
On RANDOM-5M, generated uniformly at random, the uniform distribution yields very few common non-zero dimensions between vectors, causing graph-based methods (PYANNS, HNSW, SHNSW) to suffer severe connectivity loss and low recall. The clustering effectiveness of SEISMIC is also sensitive to data distributions, 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.

2) Index Size and Construction Time: Figure 9 summarizes the index size (all including datasets and search index storage) 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 time. SEISMIC, which stores summary vectors for each block, yields the largest index size; on AntSparse-10M, its size is 3.8× 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× that of SINDI on SPLADE-FULL. In comparison, SINDI mainly sorts each vector's non-zero entries for pruning, keeping the computational overhead low and enabling rapid index building.

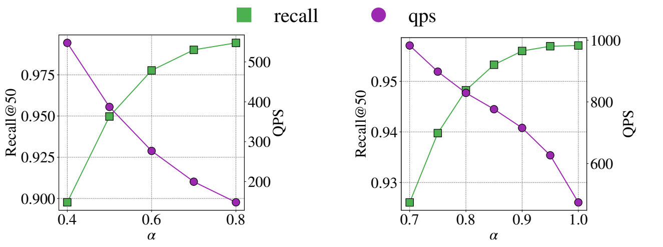
C. Parameters

1) The Impact of α: This experiment examines how the document pruning parameter α, which controls the proportion of high-mass non-zero entries retained, affects SINDI's performance. A larger α retains more non-zero entries per vector,



(a) SPLADE-FULL (b) AntSparse-10M

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



(a) SPLADE-FULL Recall@50 (b) AntSparse-10M Recall@50

Fig. 10: The Impact of α.

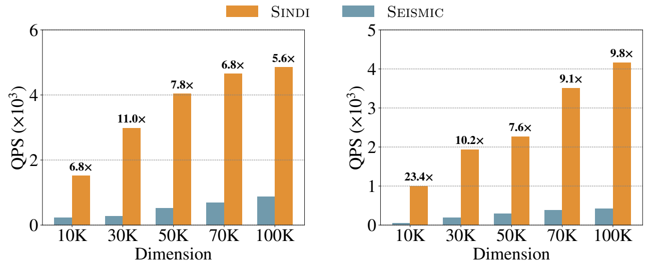
potentially increasing recall but also raising the search cost. We vary α from 0.4 to 0.8 (step 0.1) on SPLADE-FULL, and from 0.7 to 1.0 (step 0.05) on AntSparse, keeping β and γ fixed.

Figure 10 shows that, on MSMARCO, recall rises and QPS drops as α increases, with both trends flattening at higher α. In the lower α 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 α, but QPS declines more steeply.

The slower recall gain at high α is due to the *saturation effect* in Section IV-A: 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 α increase, and thus more postings to scan.

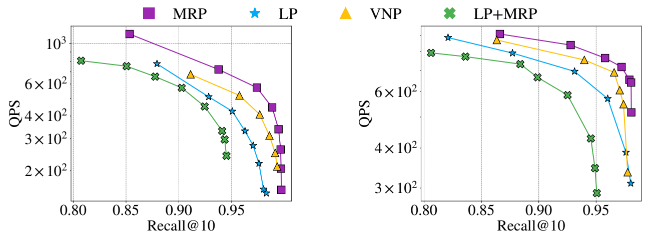
2) The Impact of sparsity: We evaluate the performance of SINDI under varying dataset *sparsity* levels. For a fixed avg $\|\vec{x}\|$, larger d yields higher *sparsity*. To examine its impact, we generate five synthetic datasets ($\|\mathcal{D}\| = $ 1M, avg $\|\vec{x}\| =$ 120) with $d \in \{$10k, 30k, 50k, 70k, 100k$\}$, 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 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.



(a) Recall@50=90% (b) Recall@50=99%

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



(a) SPLADE-FULL Recall@10 (b) AntSparse Recall@10

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

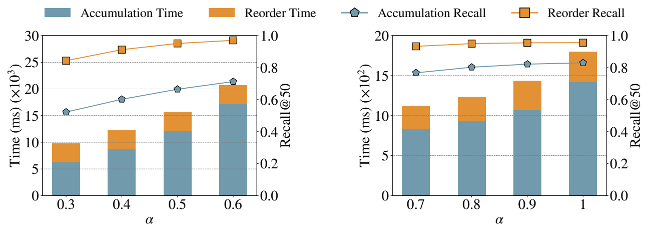
SINDI consistently outperforms SEISMIC by maintaining approximately 10× higher QPS across all *sparsity* levels. This demonstrates SINDI's efficiency and robustness to different data distributions.

D. Ablation

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 β and γ settings, while varying α 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*.

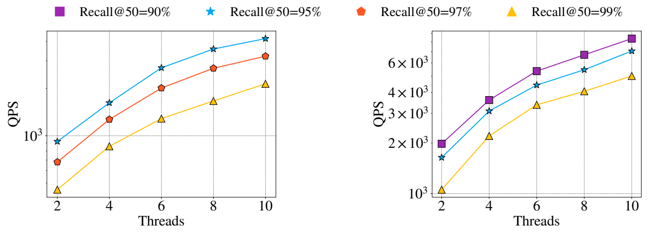
The advantage of *Mass Ratio Pruning* lies in its ability to preserve the non-zero entries that contribute most to the inner product, thereby retaining more true nearest neighbors during the coarse recall stage. In contrast, *List Pruning* limits the posting list size for each dimension, which can result 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.

2) The Impact of Reorder: To investigate the effectiveness of the reordering strategy, we evaluated SINDI's Recall@50 and query time with and without reordering on the SPLADE-FULL and AntSparse-10M datasets. Regarding parameter selection, β and γ were fixed; these values serve as representative samples drawn from the optimal intervals identified via grid



(a) SPLADE-FULL (b) AntSparse-10M

Fig. 13: Reorder vs. Non-Reorder on SPLADE-FULL and AntSparse-10M Datasets: Time Cost and Recall@50 with Varying α.



(a) SPLADE-FULL (b) AntSparse-10M

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

search. Conversely, to assess robustness across varying index densities, we varied the document prune ratio α from 0.3 to 0.6 on SPLADE-FULL and from 0.7 to 1.0 on AntSparse-10M.

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 γ is fixed, the reorder time remains relatively constant. As α 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 γ is sufficient to contain the true nearest neighbors, thereby improving both efficiency and accuracy.

E. 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 (≈ 989.75 QPS per core), whereas 10 cores achieve 8374.01 QPS (≈ 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 (≈ 226.73 QPS per core), whereas 10 cores achieve 2142.05 QPS (≈ 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.

VI. Related Work

Inverted Index-based Methods. Algorithms like BMW [40], BMP [25], and SEISMIC [23] utilize blocking strategies. BMW [40]'s dynamic pruning often degenerates to brute force due to smooth weight distributions, causing frequent cache misses. BMP [25] improves this by pre-sorting block bounds but incurs prohibitive sorting costs. SEISMIC [23] utilizes geometric blocking for skipping but remains bottlenecked by inefficient exact scoring and random memory access to raw vectors.

Graph-based Methods. PYANNS [26] and SHNSW [38] adapt HNSW [37] for sparse data via quantization and data co-location. However, performance degrades on highly sparse datasets where limited node-query overlap causes greedy routing failures. Furthermore, graph index construction remains significantly more expensive than inverted indices.

Hashing-based Methods. SINNAMON [39] and SOSIA [24] explore hashing pathways. SINNAMON [39] uses dense sketches for SIMD streaming but lacks static pruning, forcing the evaluation of extensive candidates. SOSIA [24] employs MinHash for LSH-based estimation; achieving high accuracy requires numerous hash functions, incurring significant overhead, while its randomized storage forces expensive random memory access.

VII. 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.

VIII. AI-Generated Content Acknowledgement

No content of this paper is generated by Generative AI tools and technologies, such as ChatGPT.

References

[1] K. Sawarkar, A. Mangal, and S. R. Solanki, "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, 2024, pp. 155–161.

[2] T. Şakar and H. Emekci, "Maximizing rag efficiency: A comparative analysis of rag methods," *Natural Language Processing*, vol. 31, no. 1, pp. 1–25, 2025.

[3] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Advances in Neural Information Processing Systems*, vol. 33. Curran Associates, Inc., 2020, pp. 9459–9474. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf

[4] W. Su, Y. Tang, Q. Ai, J. Yan, C. Wang, H. Wang, Z. Ye, Y. Zhou, and Y. Liu, "Parametric retrieval augmented generation," in *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '25. New York, NY, USA: Association for Computing Machinery, 2025, pp. 1240–1250. [Online]. Available: https://doi.org/10.1145/3726302.3729957

[5] D. Quinn, M. Nouri, N. Patel, J. Salihu, A. Salemi, S. Lee, H. Zamani, and M. Alian, "Accelerating retrieval-augmented generation," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, pp. 15–32. [Online]. Available: https://doi.org/10.1145/3669940.3707264

[6] I. Pogrebinsky, D. Carmel, and O. Kurland, "Enhancing retrieval-augmented generation for text completion through query selection," in *Proceedings of the ACM International Conference on the Theory of Information Retrieval*, ser. ICTIR '25. New York, NY, USA: Association for Computing Machinery, 2025, pp. 410–415. [Online]. Available: https://doi.org/10.1145/3731120.3744610

[7] G. Ma, Y. Ma, X. Gou, Z. Su, M. Zhou, and S. Hu, "Lightretriever: A LLM-based hybrid retrieval architecture with 1000x faster query inference," arXiv:2505.12260, 2025. [Online]. Available: https://arxiv.org/abs/2505.12260

[8] A. Sallinen, S. Krsteski, P. Teiletche, M.-A. Allard, B. Lecoeur, M. Zhang, F. Nemo, D. Kalajdzic, M. Meyer, and M.-A. Hartley, "Mmore: Massive multimodal open rag & extraction," 2025. [Online]. Available: https://arxiv.org/abs/2509.11937

[9] M. A. Ahmad, M. Ballout, R. A. Ahmad, and E. Bruni, "Transformer tafsir at qias 2025 shared task: Hybrid retrieval-augmented generation for islamic knowledge question answering," 2025. [Online]. Available: https://arxiv.org/abs/2509.23793

[10] C. Fensore, K. Dhole, J. C. Ho, and E. Agichtein, "Evaluating hybrid retrieval augmented generation using dynamic test sets: Liverag challenge," 2025. [Online]. Available: https://arxiv.org/abs/2506.22644

[11] T. Formal, C. Lassance, B. Piwowarski, and S. Clinchant, "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*, ser. SIGIR '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 2353–2359. [Online]. Available: https://doi.org/10.1145/3477495.3531857

[12] ——, "Towards effective and efficient sparse neural information retrieval," *ACM Transactions on Information Systems*, vol. 42, no. 5, Apr. 2024. [Online]. Available: https://doi.org/10.1145/3634912

[13] T. Formal, B. Piwowarski, and S. Clinchant, "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*, ser. SIGIR '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 2288–2292. [Online]. Available: https://doi.org/10.1145/3404835.3463098

[14] C. Lassance and S. Clinchant, "An efficiency study for SPLADE models," in *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 2220–2226. [Online]. Available: https://doi.org/10.1145/3477495.3531833

[15] O. Keivani, K. Sinha, and P. Ram, "Improved maximum inner product search with better theoretical guarantee using randomized partition trees," *Machine Learning*, vol. 107, no. 6, pp. 1069–1094, Jun. 2018. [Online]. Available: https://doi.org/10.1007/s10994-018-5711-7

[16] N. Pham, "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*, ser. KDD '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 1339–1347. [Online]. Available: https://doi.org/10.1145/3447548.3467345

[17] Y. Song, Y. Gu, R. Zhang, and G. Yu, "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, 2021, pp. 1619–1630.

[18] X. Yan, J. Li, X. Dai, H. Chen, and J. Cheng, "Norm-ranging LSH for maximum inner product search," in *Advances in Neural Information Processing Systems*, vol. 31. Curran Associates, Inc., 2018. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2018/file/b60c5ab647a27045b462934977ccad9a-Paper.pdf

[19] X. Zhao, B. Zheng, X. Yi, X. Luan, C. Xie, X. Zhou, and C. S. Jensen, "FARGO: Fast maximum inner product search via global multi-probing," *Proceedings of the VLDB Endowment*, vol. 16, no. 5, pp. 1100–1112, Jan. 2023. [Online]. Available: https://doi.org/10.14778/3579075.3579084

[20] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, 1998, pp. 604–613.

[21] F. Abuzaid, G. Sethi, P. Bailis, and M. Zaharia, "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, 2019, pp. 1250–1261.

[22] B. Neyshabur and N. Srebro, "On symmetric and asymmetric LSHs for inner product search," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML '15. JMLR.org, 2015, pp. 1926–1934.

[23] S. Bruch, F. M. Nardini, C. Rulli, and R. Venturini, "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*, 2024, pp. 152–162.

[24] X. Zhao, Z. Chen, K. Huang, R. Zhang, B. Zheng, and X. Zhou, "Efficient approximate maximum inner product search over sparse vectors," in *Proceedings of the 2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 2024, pp. 3961–3974.

[25] A. Mallia, T. Suel, and N. Tonellotto, "Faster learned sparse retrieval with block-max pruning," in *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 2411–2415. [Online]. Available: https://doi.org/10.1145/3626772.3657906

[26] PyANNS Contributors, "Pyanns: C++ code for approximate nearest neighbor search," 2025, gitHub repository. [Online]. Available: https://github.com/veaaaab/pyanns

[27] Linux Kernel Organization, "perf: Linux profiling with performance counters," 2025, online; accessed 10 June 2025. [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

[28] Intel Corporation, "Intel vtune profiler," 2025, online; accessed 10 June 2025. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html

[29] D. Bertsekas and R. Gallager, *Data Networks*. Athena Scientific, 2021.

[30] J. L. Hennessy and D. A. Patterson, *Computer architecture: A quantitative approach*. Elsevier, 2011.

[31] N. R. Draper, *Applied regression analysis*. McGraw-Hill Inc., 1998.

[32] C. L. Lawson and R. J. Hanson, *Solving least squares problems*. SIAM, 1995.

[33] T. Nguyen, M. Rosenberg, X. Song, J. Gao, S. Tiwary, R. Majumder, and L. Deng, "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)*, ser. CEUR Workshop Proceedings, vol. 1773. CEUR-WS.org, 2016. [Online]. Available: https://ceur-ws.org/Vol-1773/CoCoNIPS_2016_paper9.pdf

[34] T. Kwiatkowski, J. Palomaki, O. Redfield, M. Collins, A. Parikh, C. Alberti, D. Epstein, I. Polosukhin, J. Devlin, K. Lee, K. Toutanova, L. Jones, M. Kelcey, M.-W. Chang, A. M. Dai, J. Uszkoreit, Q. Le, and S. Petrov, "Natural questions: A benchmark for question answering research," *Transactions of the Association for Computational Linguistics*, vol. 7, pp. 452–466, 2019. [Online]. Available: https://aclanthology.org/Q19-1026/

[35] J. Chen, S. Xiao, P. Zhang, K. Luo, D. Lian, and Z. Liu, "M3-embedding: Multi-linguality, multi-functionality, multi-granularity text embeddings through self-knowledge distillation," in *Findings of the Association for Computational Linguistics: ACL 2024*. Association for Computational Linguistics, Aug. 2024, pp. 2318–2335. [Online]. Available: https://aclanthology.org/2024.findings-acl.137/

[36] P. Carlson, W. Xie, S. He, and T. Yang, "Dynamic superblock pruning for fast learned sparse retrieval," in *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '25. New York, NY, USA: Association for Computing Machinery, 2025, pp. 3004–3009. [Online]. Available: https://doi.org/10.1145/3726302.3730183

[37] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE transactions on pattern analysis and machine intelligence*, vol. 42, no. 4, pp. 824–836, 2018.

[38] GrassRMA Contributors, "Shnsw: C++ code for approximate nearest neighbor search," 2025, gitHub repository. [Online]. Available: https://github.com/Leslie-Chung/GrassRMA

[39] S. Bruch, F. M. Nardini, A. Ingber, and E. Liberty, "An approximate algorithm for maximum inner product search over streaming sparse vectors," *ACM Transactions on Information Systems*, vol. 42, no. 2, pp. 1–43, 2023.

[40] S. Ding and T. Suel, "Faster top-k document retrieval using block-max indexes," in *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 993–1002. [Online]. Available: https://doi.org/10.1145/2009916.2010048

[41] J. Zhang, Z. Shen, S. Yang, L. Meng, C. Xiao, W. Jia, Y. Li, Q. Sun, W. Zhang, and X. Lin, "High-ratio compression for machine-generated data," 2023. [Online]. Available: https://arxiv.org/abs/2311.13947

[42] Z. Yang, C. Yang, F. Han, M. Zhuang, B. Yang, Z. Yang, X. Cheng, Y. Zhao, W. Shi, H. Xi *et al.*, "Oceanbase: a 707 million tpmc distributed relational database system," *Proceedings of the VLDB Endowment*, vol. 15, no. 12, pp. 3385–3397, 2022.

Appendix

A. SIMD Implementation and Evaluation

SINDI's handwritten AVX-512 implementation relies on four key strategies:

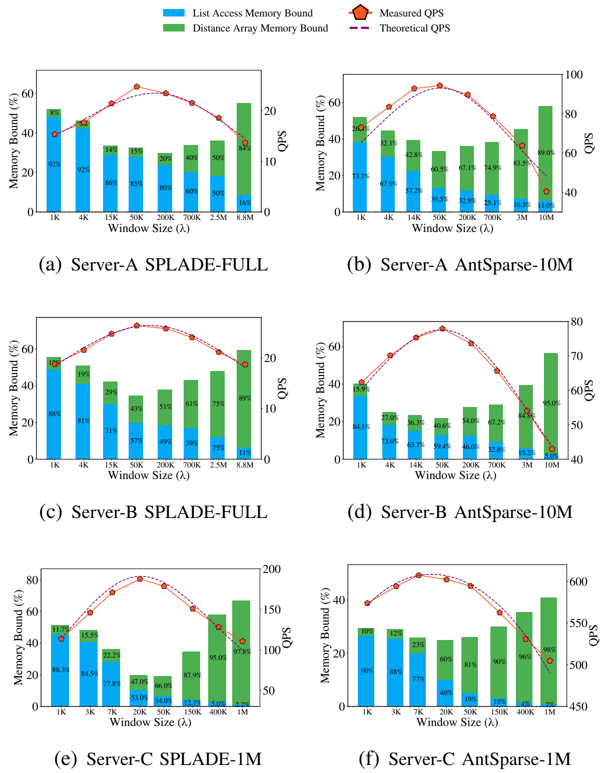
- **Latency Hiding for Sparse Access:** SINDI extend SIMD to sparse data using `gather/scatter` instructions; to hide their significant memory access latency, SINDI employ `2x loop unrolling` to schedule independent gather operations back-to-back.
- **Data Prefetching:** SINDI employ `prefetch` directives to proactively hint the CPU to fetch data for subsequent loop iterations into the L1 cache.
- **Fused Multiply-Add (FMA):** The core computation utilizes FMA instructions, which perform multiplication and addition in a single, high-throughput operation.
- **Tail Masking Process:** SINDI uses AVX-512's mask registers to efficiently process the final few elements of a list that do not fill a full SIMD vector.

To evaluate effect of SIMD to SINDI, a comparative analysis was performed between handwritten SIMD implementations and naive C++ code compiled with auto-vectorization. Experiments were conducted on an Intel Xeon Platinum 8269CY CPU using GCC 10.2.1. Tested configurations include:

- **unoptimized**: Naive C++ code without compiler optimizations.
- **compiler-opt**: Naive C++ code compiled with auto-vectorization and optimization flags (`-Ofast`, `-O3`, `-free-vectorize`, `-fopen-simd`, `-funroll-loops`).
- **sse**: Handwritten code using the respective SIMD intrinsics "`-msse -msse2 -msse3 -mssse3 -msse4 -msse4a -msse4.1 -msse4.2`".
- **avx**: Handwritten code using the respective SIMD intrinsics "`-mavx`".
- **avx2**: Handwritten code using the respective SIMD intrinsics "`-mavx2 -mfma`".
- **avx512**: Handwritten code using the respective SIMD intrinsics "`-mavx512f -mavx512pf -mavx512er -mavx512cd -mavx512vl -mavx512bw -mavx512dq -mavx512ifma -mavx512vbmi`".

The end-to-end QPS results on the SPLADE-1M and SPLADE-FULL datasets are presented in Table IV and Table V.

Results indicate that handwritten AVX-512 consistently yields peak performance (up to 1.32× speedup), validating the low-level optimizations. Notably, the compiler-optimized version (`compiler-opt`) achieves efficiency comparable to AVX-512 (e.g., 1.22× vs. 1.26× on SPLADE-1M). This confirms that the memory design of SINDI is inherently amenable to vectorization, allowing efficient deployment on platforms lacking AVX-512 via standard compiler flags.



(a) Server-A SPLADE-FULL

(b) Server-A AntSparse-10M

(c) Server-B SPLADE-FULL

(d) Server-B AntSparse-10M

(e) Server-C SPLADE-1M

(f) Server-C AntSparse-1M

Fig. 15: Impact of Window Size on Query Throughput and Memory Accesses across different hardware configurations.

B. Sensitivity Analysis of Window Size λ across Hardware and Datasets

To assess the portability of the window size parameter λ and validate the robustness of the proposed double power-law cost model, we reproduced the sensitivity analysis (originally Example 6) across three distinct server configurations (specifications detailed in Table VI). Experiments were conducted on both SPLADE (English) and AntSparse (Chinese) datasets using full-precision SINDI. We utilized Intel VTune Profiler to measure memory-bound metrics, specifically distinguishing between random accesses to the distance array and cache eviction overheads during sub-list switching. For each dataset $\mathcal{D}$, we sampled 8 values of λ on a logarithmic scale ranging from $1K$ to $||\mathcal{D}||$.

Figure 15 illustrates the correlation between theoretical predictions and measured QPS, alongside the memory-bound proportions. Table VII summarizes the fitted theoretical optima (λ^*), the measured optimal intervals, and the corresponding QPS stability ranges. Due to memory limitations, Server C can only test datasets with a size of 1M.

Analysis of Influencing Factors. The optimal window size λ^* is primarily determined by two factors: hardware cache capacity and dataset sparsity.

- **Cache Capacity:** Larger L3 caches accommodate larger distance arrays, reducing random access penalties (c_{rand}) and shifting the equilibrium λ^* to higher values. For instance, Server A (36MB L3) exhibits a larger λ^* than Server C (20MB L3) on the SPLADE dataset.

λ	unoptimized	sse	avx	avx2	compiler-opt	avx512
100000	1407.3	1413.6 (1.00×)	1513.1 (1.08×)	1584.2 (1.13×)	1711.1 (1.22×)	**1776.4 (1.26×)**
300000	1141.6	1153.1 (1.01×)	1276.7 (1.12×)	1335.3 (1.17×)	1439.0 (1.26×)	**1482.4 (1.30×)**
500000	983.2	1010.6 (1.03×)	1104.2 (1.12×)	1183.6 (1.20×)	1240.1 (1.26×)	**1275.9 (1.30×)**
700000	909.5	957.7 (1.05×)	1024.8 (1.13×)	1085.5 (1.19×)	1182.3 (1.30×)	**1203.4 (1.32×)**
1000000	828.9	858.9 (1.04×)	936.2 (1.13×)	975.2 (1.18×)	1060.8 (1.28×)	**1071.3 (1.29×)**

TABLE IV: *QPS and Speedup relative to scalar baseline on SPLADE-1M dataset.*

λ	unoptimized	sse	avx	avx2	compiler-opt	avx512
100000	276.9	282.5 (1.02×)	298.5 (1.08×)	318.8 (1.15×)	339.1 (1.22×)	**348.3 (1.26×)**
300000	218.6	228.7 (1.05×)	242.7 (1.11×)	252.3 (1.15×)	257.4 (1.18×)	**268.1 (1.23×)**
500000	194.7	199.7 (1.03×)	206.3 (1.06×)	215.9 (1.11×)	220.6 (1.13×)	**238.8 (1.23×)**
700000	182.5	186.7 (1.02×)	194.8 (1.07×)	198.6 (1.09×)	206.7 (1.13×)	**220.8 (1.21×)**
1000000	172.4	178.5 (1.04×)	181.3 (1.05×)	184.1 (1.07×)	192.0 (1.11×)	**205.8 (1.19×)**

TABLE V: *QPS and Speedup relative to scalar baseline on SPLADE-FULL dataset.*

TABLE VI: Hardware Specifications for Sensitivity Analysis

Component	**Server A**	**Server B**	**Server C**
CPU	Intel Xeon Platinum 8269CY @ 2.50GHz	Intel Xeon Platinum 8163 @ 2.50GHz	Intel Xeon CPU E5-2650 v2 @ 2.60GHz
L1d Cache	32K	32K	32K
L1i Cache	32K	32K	32K
L2 Cache	1,024K	1,024K	256K
L3 Cache	36,608K	33,792K	20,480K
Memory	512 GB	502 GB	125 GB

TABLE VII: Theoretical λ^* and Optimal Window Interval with corresponding QPS Interval

Dataset	**Server**	λ^*	λ **interval**	**QPS interval**
SPLADE-FULL	Server A	120,656	[15k, 200k]	[21, 24]
AntSparse-10M	Server A	51,024	[14k, 200k]	[90, 94]
SPLADE-FULL	Server B	92,936	[15k, 200k]	[25, 26]
AntSparse-10M	Server B	47,555	[14k, 200k]	[74, 78]
SPLADE-1M	Server C	20,794	[7k, 50k]	[171, 187]
AntSparse-1M	Server C	11,308	[7k, 50k]	[594, 607]

- **Dataset Sparsity:** Higher sparsity (as seen in AntSparse) results in shorter inverted lists, lowering the memory bandwidth cost of switching windows (c_{evict}). Consequently, highly sparse data favors smaller optimal windows. Table VII consistently shows smaller λ^* values for AntSparse compared to SPLADE across all servers.

Practical Parameter Recommendations. Our analysis reveals high parameter robustness, with query throughput remaining stable across order-of-magnitude ranges. Based on these findings, we recommend a simple **binary search method** to efficiently locate the extremum, although a general "rule of thumb" range (e.g., $\lambda \in [10,000, 120,000]$) is usually sufficient due to the flat performance plateau. For sparser datasets or cache-limited server, we recommend $\lambda \in [10,000, 50,000]$; for denser datasets or cache-rich server, we recommend $\lambda \in [50,000, 120,000]$.

C. Baseline Parameter Optimization and Configuration

To ensure the fairness and optimality of the comparative evaluation, SINDI adopted a hybrid parameter selection strategy derived from three sources: recommendations from original publications, default settings from recognized benchmarks (e.g., BigANN), and extensive self-tuning via grid search. Table VIII summarizes the final parameter settings used for all algorithms across datasets, while Table IX details the specific methodology and search ranges employed. PYANNS and HGRAPH's performance under RANDOM dataset is very poor due to random dataset's query and base vector's inner product almost zero.

The selection logic is threefold:

- **Literature Recommendations:** For algorithms such as SEISMIC and SOSIA, we adopted the optimal parameters recommended in their respective original papers, as these settings were derived from extensive grid searches.
- **Benchmark Defaults:** For PYANNS, we utilized the configuration from the BigANN Benchmark, where it secured the top position in the Sparse Track. These settings represent the current state-of-the-art standard for SPLADE-based retrieval.
- **High-Recall Optimization:** For scenarios without established defaults (e.g., BMP or SEISMIC on the new AntSparse datasets), we performed grid searches specifically optimizing for Query Per Second (QPS) throughput within the high-recall region (Recall@50 $\in [0.9, 1.0]$).

D. Comparison with Exact Retrieval Baseline (BMW)

Block-Max WAND (BMW) is a classic dynamic pruning algorithm designed for efficient exact retrieval on full-text search. It relies on storing upper-bound scores for blocks of postings to skip non-competitive documents. To validate the efficiency of SINDI's hardware-aware design against this algorithmic skipping approach, we compared the query throughput

TABLE VIII: Final Parameter Settings for All Algorithms and Datasets

Dataset	SEISMIC (λ, β, α)	PYANNS	SOSIA	BMP	HNSW
SPLADE-1M	$\lambda = 1400, \beta = 140, \alpha = 0.4$	$R = 32$ $L = 1000$	$l = 50$ $m = 150$	$b = 16$	$R = 32$ $L = 1000$
SPLADE-FULL	$\lambda = 6000, \beta = 400, \alpha = 0.4$				
NQ	$\lambda = 5250, \beta = 525, \alpha = 0.5$				
AntSparse-1M	$\lambda = 2000, \beta = 200, \alpha = 0.5$				
AntSparse-10M	$\lambda = 10000, \beta = 1000, \alpha = 0.5$				
RANDOM-5M	$\lambda = 6000, \beta = 600, \alpha = 0.4$	–			–

TABLE IX: Parameter Selection Methodology and Tuning Ranges

Algorithm	Dataset	Methodology & Search Ranges
SEISMIC	SPLADE-1M	**Self-Tuning:** Grid search $\lambda \in [1000, 2000]$ (step 200).
	SPLADE-FULL	**Paper Rec.** : Grid search $\lambda \in [1500, 7500]$ (step 500), $\beta \in [150, 750]$ (step 50), $\alpha \in [0.1, 0.5]$ (step 0.1).
	NQ	**Paper Rec.** : Grid search $\lambda \in \{4500, ...\}$, $\beta \in \{300, ...\}$, $\alpha \in \{0.3, 0.4, 0.5\}$.
	RANDOM-5M	**Self-Tuning:** Grid search $\lambda \in [3000, 10000]$ (step 1000).
	AntSparse-1M	**Self-Tuning:** Grid search $\lambda \in [800, 2000]$ (step 200).
	AntSparse-10M	**Self-Tuning:** Grid search $\lambda \in [2000, 10000]$ (step 1000).
PYANNS	SPLADE	**Benchmark Rec.** : BigANN Sparse Track winner config.
	Others	**Self-Tuning:** Grid search $R \in \{16, 32, 64\}, L \in \{1000, 2000\}$.
SOSIA	All	**Paper Rec.** : Grid search $l \in \{1, 10, 20, 40, 80\}$, $m \in \{25, 50, 100, 150, 200\}$.
BMP	All	**Self-Tuning:** Grid search $b \in \{16, 32, 64\}$.
HNSW	All	**Default:** Inherits optimal settings from PYANNS.

(QPS) of BMW using the PISA engine against the full-precision version of SINDI using the PISA engine. The results for Top-50 and Top-100 retrieval are presented in Table X.

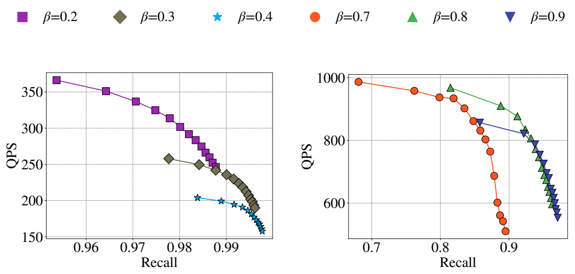
TABLE X: Performance Comparison (QPS) of BMW and Full-Precision SINDI

	Top-50		Top-100	
Dataset	**BMW**	**SINDI**	**BMW**	**SINDI**
SPLADE-1M	5.22	**301.30**	5.33	**304.06**
SPLADE-FULL	0.68	**33.78**	0.66	**32.85**
AntSparse-1M	436.55	**1029.59**	473.20	**1022.98**
AntSparse-10M	75.23	**104.56**	66.98	**103.36**
RANDOM-5M	6.93	**124.57**	7.19	**127.12**
NQ	1.72	**90.89**	1.67	**90.85**

Analysis. SINDI significantly outperforms BMW on SPLADE datasets (approx. 50× speedup). This is because SPLADE vectors exhibit smooth score distributions that render BMW's block-skipping ineffective. Moreover, SPLADE's long queries (nearly 50) leads to many branch prediction errors and cache misses. However, on the AntSparse dataset, the performance gap narrows. AntSparse is characterized by extremely high dimensionality (250k) and very short queries (avg. 6 terms). In this sparse regime, BMW's ability to skip posting lists becomes more advantageous. SINDI maintains superior query throughput due to its cache-friendly memory layout and lack of branch misprediction overhead.

E. Sensitivity Analysis of Reordering Depth γ

To evaluate the sensitivity of retrieval performance to the reordering candidate size γ, experiments were conducted on the **SPLADE-FULL** and **AntSparse-10M** datasets. The parameter γ was varied across broad ranges under three distinct query pruning ratios (β). Figure 16 illustrates the resulting QPS-Recall trajectories.



(a) SPLADE-FULL (b) AntSparse-10M

Fig. 16: Sensitivity Analysis of Reordering Candidate Size γ.

Analysis. The curves exhibit a consistent trend across all configurations: recall increases with γ and eventually reaches a saturation plateau. It is observed that this saturation occurs more rapidly as the query pruning ratio β increases. Extending γ beyond the saturation point offers negligible improvements in recall but causes a linear degradation in QPS due to the increased cost of full-precision computations.

F. Performance Comparison with Lucene-based System

To demonstrate the fundamental advantages of SINDI over standard "IR-style" indexing, a system-level comparison was conducted against Elasticsearch, the most widely adopted production system based on the Lucene core. Evaluations were performed in two distinct scenarios to ensure robustness: a standalone system comparison and an industrial integration assessment.

1) **Standalone System Comparison:** SINDI (integrated into TBaseSearch [41]) was compared against Elasticsearch under identical configurations (4 threads, top-$k = 10$). SINDI achieved **1,775 QPS**, approximately **3× higher** than Elasticsearch's 575 QPS, while concurrently reducing average latency by **67%** (2.23 ms vs. 6.92 ms). These metrics confirm the superior core efficiency of the proposed architecture.
2) **Industrial Integration (OceanBase):** To validate performance within complex commercial infrastructure, OceanBase [42] (a distributed relational database integrating SINDI) was evaluated against Elasticsearch on the SPLADE-FULL dataset using Intel Xeon Platinum 8269CY CPUs. As shown in Figure 17, the integration consistently outperforms the baseline:
   - **Full Recall (No Pruning):** OceanBase achieves **78 QPS** compared to Elasticsearch's 33 QPS, representing a **2.4×** speedup even without approximation strategies.
   - **With Pruning:** Performance gains are approximately **3×** without reordering, extending to **2×–5×** when reordering is enabled.

These results indicate that SINDI is fundamentally more efficient than standard IR-style implementations. While Lucene effectively utilizes skipping (Block-Max WAND), it remains limited by scalar processing. In contrast, the value-storing design of SINDI facilitates cache-friendly access and **SIMD acceleration**, delivering order-of-magnitude improvements over traditional methods.

TABLE XI: Performance Comparison between SINDI and Elasticsearch

Metric	**TbaseSearch**	**Elasticsearch**
Memory Usage (RSS)	1.09 GB	1.65 GB
Number of Queries	1,000	1,000
Top-k	10	10
Client Threads	4	4
Average Recall	0.9988	0.9950
QPS (Queries Per Second)	1,775.11 (**+208.4%**)	575.61
Avg Latency (ms)	2.23 (**-67.8%**)	6.92
Min Latency (ms)	0.58 (**-82.2%**)	3.26
P99 Latency (ms)	4.22 (**-63.7%**)	11.62

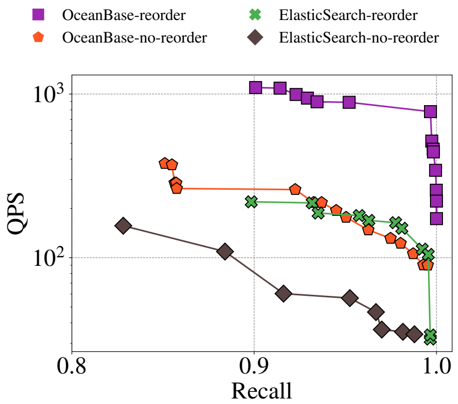


Fig. 17: Performance comparison between OceanBase (integrating SINDI) and Elasticsearch on the SPLADE-FULL dataset.