VSAG: An Optimized Search Framework for Graph-based Approximate Nearest Neighbor Search [Industry]

Xiaoyao Zhong[1], Haotian Li[1], Jiabao Jin[1], Mingyu Yang[1], Deming Chu[1], Xiangyu Wang[1],
Zhitao Shen[1], Wei Jia[1], George Gu[3], Yi Xie[3], Xuemin Lin[4],
Heng Tao Shen[2], Jingkuan Song[2], Peng Cheng[2]
[1]Ant Group, Shanghai, China; [2]Tongji University, Shanghai, China;
[3]Intel Corporation, Shanghai, China; [4]Shanghai Jiaotong University, Shanghai, China;
{zhongxiaoyao.zxy, tianlan.lht, jinjiabao.jjb, yangming.ymy, chudeming.cdm, wxy407827, zhitao.szt, jw94525}@antgroup.com;
{george.gu, ethan.xie}@intel.com; shenhengtao@hotmail.com; {xuemin.lin, jingkuan.song, chengpeng.cs}@gmail.com

ABSTRACT

Approximate nearest neighbor search (ANNS) is a fundamental problem in vector databases and AI infrastructures. Recent graph-based ANNS algorithms have achieved high search accuracy with practical efficiency. Despite the advancements, these algorithms still face performance bottlenecks in production, due to the random memory access patterns of graph-based search and the high computational overheads of vector distance. In addition, the performance of a graph-based ANNS algorithm is highly sensitive to parameters, while selecting the optimal parameters is cost-prohibitive, e.g., manual tuning requires repeatedly re-building the index.

This paper introduces *VSAG*, a framework that aims to enhance the in-production performance of graph-based ANNS algorithms. *VSAG* has been deployed at scale in the services of Ant Group, and it incorporates three key optimizations: *(i) efficient memory access*: it reduces L3 cache misses with pre-fetching and cache-friendly vector organization; *(ii) automated parameter tuning*: it automatically selects performance-optimal parameters without requiring index rebuilding; *(iii) efficient distance computation*: it leverages modern hardware, scalar quantization, and smartly switches to low-precision vector to dramatically reduce the distance computation costs. We evaluate *VSAG* with real-world datasets. The experimental results show that *VSAG* achieves the state-of-the-art performance and provides up to 4x more QPS speed up than HNSWlib (an industry-standard library) while ensuring the same accuracy.

PVLDB Reference Format:
Xiaoyao Zhong, Haotian Li, Jiabao Jin, Mingyu Yang, Deming Chu, Xiangyu Wang, Zhitao Shen, Wei Jia, George Gu, Yi Xie, Xuemin Lin, Heng Tao Shen, Jingkuan Song, Peng Cheng. VSAG: An Optimized Search Framework for Graph-based Approximate Nearest Neighbor Search [Industry]. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/antgroup/vsag.

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 this license. For any use beyond those covered by this license, obtain permission by emailing 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

1 INTRODUCTION

At Ant Group [1], we have observed an increasing demand to manage large-scale high-dimensional vectors across different departments. This demand is fueled by two factors. First, the advent of Retrieval-Augmented Generation (RAG) for large language models (LLMs) [23, 36] needs vector search to address issues such as hallucinations and outdated information. Second, the explosive growth of unstructured data (e.g., documents, images, and videos), requires efficient analysis and storage methods. Many systems transform these unstructured data into embedding vectors for efficient retrieval, e.g., Alipay's facial-recognition payment [2], Google's image search [25], and YouTube's video search [56].

Approximate nearest neighbor search (ANNS) is the foundation for these AI and LLM applications. Due to the curse of dimensionality [27], exact nearest neighbor search becomes prohibitively expensive as dimensionality grows. ANNS, however, trades off a small degree of accuracy for a substantial boost in efficiency, establishing itself as the gold standard for large-scale vector retrieval.

Recently, graph-based ANNS algorithms (e.g., HNSW [40] and VAMANA [29]) successfully balance high recall with practical runtime performance. These methods typically construct a graph, where each node is a vector and each edge connects nearby vector pairs. During a query, an approximate k-nearest neighbor search starts from a random node and greedily moves closer to the query vector x_q, thereby retrieving its k nearest neighbors.

Despite their success, existing graph-based ANNS solutions still face considerable performance challenges. First, they incur *random memory-access overhead*, since graph traversals with arbitrary jumps often lead to frequent cache misses and elevated costs. Second, repeated *distance computations* across candidate vectors can dominate total runtime, especially when vectors are high-dimensional. Finally, performance is highly *sensitive to parameter settings* (e.g., maximum node degree and candidate pool size), yet adjusting these parameters generally requires rebuilding the index, which can take hours or even days. We use a small example from our experiments to illustrate these issues:

Modern production systems [34, 50] typically employ vector quantization to reduce the distance computation cost. Therefore, we set our baseline as **HNSW** with **SQ4** quantization [59]. We conduct 1,000 vector queries on the GIST1M[1]. In what follows, we report the performance limitations of graph-based ANNS algorithms, using the experimental evidence of the baseline: *(i) high*

[1]http://corpus-texmex.irisa.fr/

Table 1: Comparison to Existing Algorithms (GIST1M).

Metric	IVFPQFS [4]	HNSW [40]	VSAG (this work)
Memory Footprint	3.8G	4.0G	4.5G
Recall@10 (QPS=2000)	84.57%	59.46%	89.80%
QPS (Recall@10=90%)	1195	511.9	2167.3
Distance Computation Cost	0.71ms	1.62ms	0.1ms
L3 Cache Miss Rate	13.98%	94.46%	39.23%
Parameter Tuning Cost	>20h	>60h	2.92h
Parameter Tuning	manual	manual	auto

memory access cost: each query needs over 2,959 random vector accesses (total 1.4 MB), causing a 67.42% L3 cache miss rate. The memory-access operations consume 63.02% of the search time. *(ii) high parameter tuning cost*: if we use the optimal parameters instead of the manually selected values, the QPS can increase from 1,530 to 2,182, by 42.6%. However, brute-force tuning of parameters takes more than 60 hours, which is prohibitively expensive. *(iii) high distance computation cost*: distance computations still take 26.12% of the search time despite using SQ4 quantization.

Contributions. This paper presents *VSAG*, a framework for enhancing the in-production efficiency of graph-based ANNS algorithms. The optimizations of *VSAG* are in three-fold. *(i) Efficient Memory Access*: during graph-based search, it pre-fetches the neighbor vectors, and creates a continuous copy of the neighbor vectors for some vertices. This cache-friendly design can reduce L3 cache misses. *(ii) Automated Parameter Tuning: VSAG* can automatically tune parameters for environment (e.g., prefetch depth), index (e.g., max degree of graph), and query (e.g., candidate size). Suppose there are 3 index parameters and 5 choices for each parameter. The tuned index parameters of *VSAG* offer similar performance to that of brute-force tuning, which needs to enumerate all 5^3 combinations of parameters leading to a total tuning time of 5^3 times of index construction time. On the contrast, the tuning cost of *VSAG* is only 2-3 times of index construction time. *(iii) Efficient Distance Computation*: *VSAG* provides various approximate distance techniques, such as scalar quantization. All distance computation is well optimized with modern hardware, and a selective re-ranking strategy is used to ensure retrieval accuracy.

Table 1 compares the performance of *VSAG* with existing works on GIST1M. The results show that *VSAG* alleviates the performance challenges in memory access, parameter tuning, and distance computation, thus providing higher QPS with the same recall rate.

In summary, we make the following contributions:

1. We enhance the memory access of *VSAG* in §3. The L3 cache miss rate of *VSAG* is much less than that of other graph-based ANNS works.
2. We propose automatic parameter tuning for *VSAG* in §4. It automatically selects performance-optimal parameters that are comparable to grid search without requiring index rebuilding.
3. We accelerate *VSAG* in distance computation in §5. Compared with other graph-based ANNS works, *VSAG* requires much less time for distance computation.
4. We evaluate algorithms on real datasets in §6. The results show that *VSAG* can achieve the SOTA performance and outperform HNSWlib by up to 4x in QPS under the same recall guarantee.

Table 2: Symbols and Descriptions.

Symbol	Description
D	the base dataset
G	the graph index
L	the labels of edges
τ_l, τ_h	the distance function with low precision and high precision
x, x_b, x_q, x_n	a normal, base, query, and neighbor vector
$NN_k(x), ANN_k(x)$	the k nearest and approximate k nearest neighbors of x
ef_s, ef_c	the candidate pool size in search and construction phase
α_s, α_c	the pruning rate used in search and construction phase
m_s, m_c	the maximum degree of graph used in search and construction phase
ω	prefetch stride
ν	prefetch depth
δ	redundancy ratio

2 OVERVIEW OF VSAG FRAMEWORK

This section presents an overview of our *VSAG* framework, including memory access optimization, automatic parameter tuning, and distance computation acceleration. As shown in Figure 1, *VSAG* integrate those optimizations into different search phases.

We list the symbols used in this paper in Table 2.

2.1 Memory Access Optimization

Deterministic Access. Distance computations for neighboring vectors often incur random memory access patterns in graph-based algorithms, leading to significant cache misses. *VSAG* addresses this by integrating software prefetching [6] (i.e., `_mm_prefetch`) through a *Deterministic Access* strategy (see §3.2.1). *VSAG* strategically inserts prefetch instructions during and before critical computations, proactively loading target data into L3/higher-level caches. This prefetch-pipeline overlap ensures data availability before subsequent computation phases begin, effectively mitigating cache miss penalties. Furthermore, the *VSAG* framework effectively mitigates suboptimal prefetch operations through batch processing and reordering of the access sequence.

PRS. To better optimize memory access, *VSAG* introduces a *Partial Redundant Storage (PRS)* design (see §3.3), which provides a flexible and high-performance storage foundation to optimize both distance computations and memory access while balancing storage and computational resource usage. In production environments constrained by fixed hardware configurations, such as 4C16G (i.e., equipped with 4 CPU cores and 16GB of memory) and 2C8G (i.e., equipped with 2 CPU cores and 8GB of memory), most algorithms frequently exhibit resource utilization imbalances between computational and memory subsystems. During computational processes, CPUs frequently encounter idle cycles caused by cache misses, which hinders their ability to achieve optimal utilization, and thereby limits the system's QPS.

To address this challenge, the PRS framework *Redundantly Storing Vectors* (see §3.3.2) that embeds compressed neighbor vectors at each graph node. This architectural design enables batched distance computations while leveraging more efficient *Hardware-based Prefetch* [11] (see §3.3.1) to maintain high cache hit rates. By incorporating advanced quantization methods [22, 31], PRS achieves high vector storage compression ratios, thereby maintaining acceptable storage overhead despite data redundancy.

In particular, the system has a parameter called the redundancy ratio δ, providing flexible control over the *Balance of Computational Efficiency and Memory Utilization* (see §3.3.3). In compute-bound scenarios with high-throughput demands, *VSAG* adaptively

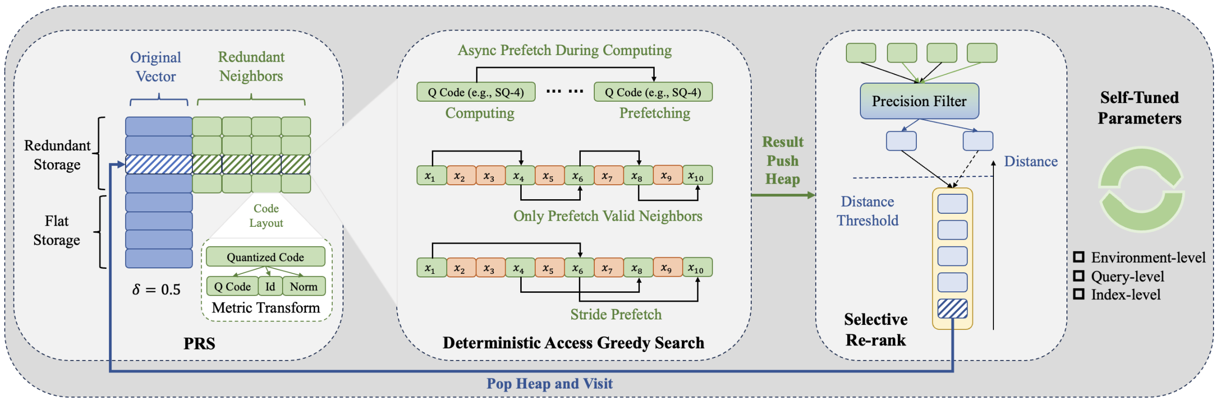


Figure 1: The Search Framework of *VSAG*.

increases the redundancy ratio to mitigate cache contention, thus minimizing CPU idle cycles during memory access while preserving storage efficiency. In contrast, in memory-constrained low-throughput scenarios, the framework strategically reduces redundancy ratio to optimize the index-memory footprint. Under memory-constrained conditions, this optimization enables deployment on reduced instance tiers, thereby curtailing compute wastage.

2.2 Automatic Parameter Tuning

VSAG addresses parameter selection complexity through a tripartite classification system with specialized optimization strategies: *environment-level*, *query-level*, and *index-level* parameters. Environment-level parameters (e.g., prefetch stride ω) exclusively influence query-per-second (QPS) performance without recall rate impacts, thus incurring the lowest tuning overhead. Query-level parameters (e.g., candidate set size ef_s [41]) exhibit moderate tuning costs by jointly affecting QPS and recall, requiring adjustment based on query vector distributions. Index-level parameters (e.g., maximum degree m_c [41]) demand the highest tuning investment due to their tripartite impact on QPS, recall, and index construction time – parameter validation necessitates multiple index rebuilds.

- *Environment-level Parameters* (see §4.2): *VSAG* employs a grid search to identify optimal configurations for peak QPS performance through systematic parameter space exploration.
- *Query-level Parameters* (see §4.3): *VSAG* implements multi-granular tuning strategies, including *fine-grained adaptive optimization* that dynamically adjusts parameters based on real-time query difficulty assessments.
- *Index-level Parameters* (see §4.4): *VSAG* introduces a novel mask-based index compression technique that encodes multiple parameter configurations into an unified index structure. During searches, edge-label filtering dynamically emulates various construction parameters, thereby reducing index-level parameters to query-level equivalents while keeping a single physical index.

2.3 Distance Computation Optimization

Distance computation is a main overhead in vector retrieval, and its cost increases significantly with the growth of vector dimensions. *Quantization methods* (see §5.2) can effectively accelerate distance computation. For example, under identical instruction set architectures, AVX512 [28] can process 4× as many INT8 data per instruction compared to FLOAT32 values. However, naive quantization approaches often result in significant accuracy degradation. *VSAG* uses *Selective Re-rank* (see §5.4) to improve efficiency without sacrificing the search accuracy. Furthermore, specific distance metrics (i.e., euclidean distance) can be strategically decomposed and precomputed, effectively reducing the number of required instructions during actual search operations.

3 MEMORY ACCESS OPTIMIZATION

Graph-based algorithms suffer from random memory access patterns that incur frequent cache misses. The fundamental strategy for mitigating cache-related latency lies in effectively utilizing vector computation intervals to prefetch upcoming memory requests into cache. In *VSAG*, three primary optimization strategies emerge for maximizing cache utilization efficiency:

- Leveraging software prefetching to improve cache hit rates.
- Optimizing search patterns to enhance the effectiveness of software prefetching.
- Optimizing the memory layout of indexes to efficiently utilize hardware prefetching.

3.1 Software-based Prefetch: Making Random Memory Accesses Logically Continuous

As shown in Figure 2, when computing vector distances, the vector is loaded sequentially from a segment of memory. The CPU fetches data from memory in units of cache lines [49]. Consequently, multiple consecutive cache fetch operations are triggered for a single distance computation.

Example 1. *Take standard 64-byte cache line architectures as an example. The 960-dimensional vector of GIST1M stored as float32 format necessitates* $960 \times 4/64 = 60$ *cache line memory transactions, demonstrating significant pressure on memory subsystems.*

This passive caching mechanism creates operational inefficiencies by extensively fetching data only upon cache misses, resulting in synchronous execution bottlenecks. As illustrated in Figure 2, the orange timeline shows how the regular ANNS algorithm serializes the computation and memory access phases: Each cache line fill (analogous to blocking I/O) stalls computation until completion. The accumulated latency from successive cache line transfers introduces a significant constant factor in complexity analysis, particularly in memory-bound scenarios with poor data locality.

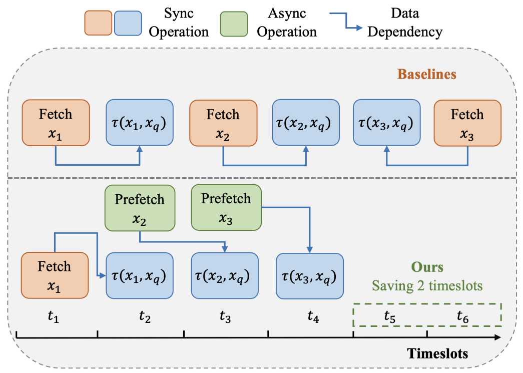


Figure 2: Passive Memory Access and Software-based Prefetch.

Software-based Prefetch. Modern CPUs support software prefetch instructions, which can asynchronously load data into different levels of the cache [6]. By leveraging the prefetch instruction to preload data, we can achieve a near-sequential memory access pattern from the CPU level. This indicates that data is preloaded into the cache before the CPU requires it, thereby preventing disruptions in the computational flow caused by random memory address loads. More specifically, prior to computing the distance for the current neighbor, the vector for the next neighbor can be prefetched. As detailed in Figure 2, the green flow represents the use of prefetching. From the perspective of the CPU, the majority of distance computations make use of data that has already been cached. Furthermore, because prefetching operates asynchronously, it does not obstruct ongoing computations. The synergy of asynchronous prefetching and immediate data access optimizes the utilization of CPU computational resources, thereby substantially enhancing search performance.

3.2 Deterministic Access Greedy Search: Advanced Prefetch Validity

In §3.1, the software-based prefetch mechanism initiates the fetching of next-neighbor vector upon completion of each neighbor computation. However, this method results in redundant operations because previously visited neighbors that do not require distance computations still generate prefetch time cost. Example 2 illustrates the inherent limitations of previous prefetch schemes:

Example 2. *In Figure 3(a), x_2 and x_3 have already been visited, and the distances do not need to be recomputed. This renders the previous prefetching of these vectors ineffective. Additionally, when computing x_4, the prefetch may also fail due to the prefetching gap being too short.*

In this section, we present two dedicated strategies to address the aforementioned prefetching challenges.

3.2.1 Deterministic Access. In contrast to prefetching during edge access checks, *VSAG* exclusively prefetches only those edges that have not been accessed. The mechanism begins by batch-processing all neighbor nodes to determine their access status. Following this, unvisited neighbors are logically grouped, and prefetching is performed collectively. This strategic approach ensures that each prefetched memory address corresponds exclusively to computation-essential data, thereby enhancing prefetching efficiency and minimizing redundant memory operations.

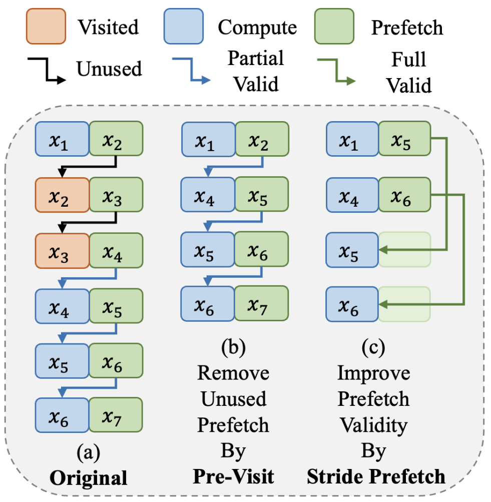


Figure 3: Three Different Prefetching Strategies

3.2.2 Stride Prefetch. Batch processing ensures that each prefetch retrieves data intended for future use. However, prefetch effectiveness varies due to the asynchronous nature of prefetching and the absence of a callback mechanism to confirm prefetch completion. Optimal performance occurs when the required data reside in the cache precisely when the computation flow demands it. Premature prefetching risks cache eviction, while delayed prefetching negates performance gains. This necessitates balancing prefetch timing with computation duration. To address this, stride prefetching dynamically aligns hardware computation throughput with software prefetching rates, maximizing prefetch utility. The key parameter, the prefetch stride ω [48], determines how many computation steps occur before each prefetch. Adjusting ω is crucial, and in §4.2, we propose an automated strategy to select its optimal value.

Example 3. *As detailed in Figure 3(b), the adjusted pattern demonstrates that during batch processing, the **Deterministic Access** strategy eliminates the need to access x_2 and x_3. Consequently, the search logic progresses from x_1 directly to x_4. This sequence modification enables the prefetch mechanism to target x_4 while computing x_1. Figure 3(c) further reveals the temporal characteristics of asynchronous prefetching: The data loading process requires two vector computation cycles to populate the cache line. When computation for x_1 initiates, only the x_4 vector can be prefetched. After the computations of x_1 and x_4, the **Stride Prefetch** strategy ensures timely cache population of x_6 data, which is immediately available for subsequent computation.*

Deterministic Access Greedy Search. The cache-optimized search algorithm is formalized in Algorithm 1. The graph index G constitutes an oriented graph that maintains base vectors along with their neighbors. The labels L of edges in G are used for automatic index-level parameters tuning (see §4). We use G_i and L_i to indicate the out-edges and labels of x_i. The low- and high-precision distance functions τ_l and τ_h are used to accelerate distance computation while maintaining search accuracy, and they are employed in the selective re-ranking process (see §5). The complete algorithm explanation is provided in Appendix A of our report [52].

Algorithm 1: Deterministic Access Greedy Search

```
Input: graph $G$, labels $L$, base dataset $D$, initial nodes $I$,
       query point $x_q$, low- and high- precision distance
       functions $\tau_l$ and $\tau_h$, search parameters $k$, $ef_s$, $m_s$, $\alpha_s$,
       $\omega$, $v$
Output: $ANN_k(x_q)$ and their high-precision distances $T$
1  candidate set $C \leftarrow$ maximum-heap with size of $ef_s$
2  visited set $V \leftarrow \emptyset$
3  insert $(x_i, \tau_l(x_i, x_q)), \forall x_i \in I$ into $C$
4  while $C$ has un-expended nodes do
5      $x_i \leftarrow$ closest un-expended nodes in $C$
6      $N \leftarrow$ empty list
7      for $j \in G_i$ do
           // Only retrieve Id(i.e., $x_j.id = j$)
8          if $j \notin V$ and $L_j \leq \alpha_s$ and $|N| < m_s$ then
9              $N \leftarrow N \cup \{j\}$
10             $V \leftarrow V \cup \{j\}$
11     for $k \in [0, min(\omega, |N|)$ do
12         prefetch $v$ cache lines start from $D_{N_k}$
13     for $k \in [0, |N|)$ do
14         if $k + \omega < |N|$ then
15             prefetch $v$ cache lines start from $D_{N_{k+\omega}}$
16         $x_j \leftarrow D_{N_k}$ // Memory Access
17         insert $(x_j, \tau_l(x_j, x_q))$ into $C$ and keep $|C| \leq ef_s$
18 $ANN_k(x_q), T \leftarrow$ selective re-rank $C$ with $\tau_l$ and $\tau_h$
19 return $ANN_k(x_q), T$
```

3.3 PRS: Flexible Storage Layout Boosting Search Performance

While incorporating well-designed prefetch patterns into search processes can theoretically improve performance, the inherent **limitations of Software-based Prefetch** prevent guaranteed memory availability for all required vectors. This phenomenon can be attributed to multiple fundamental constraints: (a) Prefetch instructions remain advisory operations rather than mandatory commands. Even when optimal prefetch patterns are implemented, their actual execution cannot be assured. (b) Cache line contention represents another critical challenge. In multi-process environments, aggressive prefetch strategies may induce L3 cache pollution through premature data loading. (c) The intrinsic cost disparity between prefetch mechanisms further compounds these issues. Software-based prefetching intrinsically carries higher operational costs and demonstrates inferior efficiency compared to hardware-implemented alternatives.

3.3.1 Hardware-based Prefetch. Hardware-based prefetching relies on hardware mechanisms that adaptively learn from cache miss events to predict memory access patterns. The system employs a training buffer that dynamically identifies recurring data access sequences to automatically prefetch anticipated data into the cache hierarchy. Compared to software-controlled prefetching, this hardware approach demonstrates better runtime efficiency while functioning transparently at the architectural level. The training mechanism shows particular effectiveness for *Sequential Memory Access* patterns [9], where it can rapidly detect and exploit sequential memory access characteristics. This optimization proves particularly beneficial for space-partitioned index structures like inverted file-based index [32], where vectors belonging to the same partition maintain contiguous storage allocation. Conversely, graph-based indexing architectures exhibit irregular access patterns with poor spatial locality, resulting in inefficient *Random Memory Access* [9]. The inherent randomness of access sequences prevents the training buffer from establishing effective pattern recognition models.

3.3.2 Redundantly Storing Vectors. VSAG integrates the benefits of space-partitioned indexes into graph-based indexing algorithms through redundant vector storage. By co-locating neighbor lists with their corresponding vectors within each node's data structure, it achieves *Continuous Uniform Address Access.* This design ensures that neighbor retrieval operations only require sequential access within contiguous memory regions, thereby fully leveraging hardware prefetching [11] capabilities.

Example 4. *As illustrated in Figure 1, consider 10 vectors stored contiguously in memory. Even when accessing x_1 through x_5 where x_2 and x_3 are not immediately required, hardware prefetchers can still proactively load x_4 into cache. This behavior stems from the memory locality created by storing adjacent vectors (x_1 to x_5) in consecutive memory addresses. The consistent memory layout and predictable access patterns effectively compensate for software-based prefetching inefficiencies through hardware optimizations.*

3.3.3 Balance of Computational Efficiency and Memory Utilization. To address the computational-memory resource imbalance caused by fixed instance specifications (e.g., 4C16G, 2C8G) in industrial applications, we propose PRS to take advantage of hardware prefetching to reduce CPU idle time. A dynamically tunable *redundancy ratio* δ controls the proportion of redundantly stored neighbor vectors in graph indices, balancing prefetch efficiency and CPU utilization. When $\delta = 1$, full redundancy maximizes hardware prefetch benefits, achieving peak CPU utilization at the cost of higher memory consumption. In contrast, $\delta = 0$ eliminates redundancy to minimize memory usage but sacrifices prefetch efficiency. This flexibility enables workload-aware resource optimization as shown in Example 5.

Example 5. *For high-throughput/high-recall scenarios (Fig. 4 (a)), increasing $\delta = 1$ prioritizes CPU efficiency to meet demanding targets with fewer compute resources. In memory-constrained environments with moderate throughput requirements (Fig. 4 (b)), reducing $\delta = 0$ alleviates memory pressure while allowing instance downsizing (e.g., 4C16G to 2C8G), which maintains service quality through controlled compute scaling and reduces infrastructure costs.*

4 AUTOMATIC PARAMETER TUNING

4.1 Parameter Classification

We observe that specific parameters in the retrieval process exert a significant influence [55] on the search performance (e.g., prefetch depth v, construction ef ef_c, search ef ef_s, and candidate pool size m_c). These parameters can be systematically categorized into three

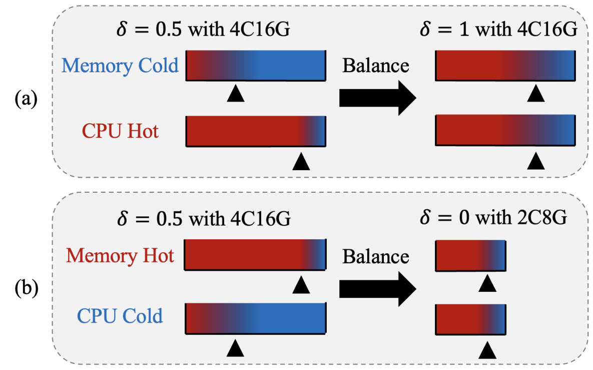


Figure 4: Adjust Redundant Ratio δ

distinct types for discussion: Environment-Level Parameters (ELPs), Query-Level Parameters (QLPs), and Index-Level Parameters (ILPs). **ELPs** primarily affect the efficiency (QPS) of the retrieval process and are directly related to the retrieval environment (e.g., prefetch stride ω, prefetch depth ν). Most of these parameters are associated with the execution speed of system instructions rather than the operational flow of the algorithm itself. For instance, prefetch-related parameters mainly influence the timing of asynchronous operations.

QLPs inherently influence both retrieval efficiency (QPS) and effectiveness (Recall) simultaneously. These parameters operate on prebuilt indexes and can be dynamically configured during the retrieval phase (e.g., search parameter ef_s, selective reranking strategies). In particular, efficiency and effectiveness exhibit an inherent trade-off: For a static index configuration, achieving higher QPS inevitably reduces Recall performance.

ILPs define an index's core structure and performance. Tuning the construction-related parameters (e.g., m_c, α_c) requires rebuilding indices. It is a process far more costly than adjusting QLPs or EIPs. Crucially, ILPs impact both efficiency and effectiveness simultaneously.

Hardness. The complexity of tuning these three parameter categories shows a progressive increase. ELPs focus solely on algorithmic efficiency, resulting in a straightforward single-objective optimization problem. In contrast, QLPs require balancing both efficiency and effectiveness criteria, thereby forming a multi-objective optimization challenge. The most demanding category, ILPs, necessitates substantial index construction time in addition to the aforementioned factors, leading to exponentially higher tuning expenditures. In subsequent sections, we present customized optimization strategies for each parameter category.

4.2 Search-based Automatic ELPs Tuner

The proposed algorithm utilizes multiple ELPs that exhibit substantial variance in optimal configurations in different testing environments and methodologies, as demonstrated by our comprehensive experimental analysis (see §6). As elaborated in §3.2.2, the stride prefetch mechanism operates through two crucial parameters: the prefetch stride ω and the prefetch depth ν. The parameter ω governs the prefetch timing based on the dynamic relationship between the CPU computational speed and the prefetch latency within specific deployment environments. In particular, smaller ω values are required to initiate earlier prefetching when faced with faster computation speeds or slower prefetch latencies. Meanwhile, the ν parameter is primarily determined by the CPU's native prefetch patterns combined with vector dimensionality characteristics.

These environment-sensitive parameters exclusively influence algorithmic efficiency metrics (QPS) while maintaining consistent effectiveness outcomes (Recall Rate), thereby enabling independent optimization distinct from core algorithmic logic. The *VSAG* tuning framework implements an optimized three-step procedure:

1. Conduct an exhaustive grid search for all combinations of environment-dependent parameter.
2. Evaluate performance metrics using statistically sampled base vectors.
3. Select parameter configurations that maximize retrieval speed while maintaining operational stability.

4.3 Fine-Grained Automatic QLPs Tuner

Observation. There are significant differences in the parameters required for different queries to achieve the target recall rate, with a highly skewed distribution. Specifically, 99% of queries can achieve the target recall rate with small query parameters, while 1% of queries require much larger parameters. Experimental results show that assigning personalized optimal retrieval parameters to each query can improve retrieval performance by 3–5 times.

To address the observation of QLPs, we propose a **Decision Model** for query difficulty classification, enabling personalized parameter tuning while maintaining computational efficiency. We introduce a learning-based adaptive parameter selection framework through a GBDT classifier [10] with early termination capabilities, demonstrating superior performance in fixed-recall scenarios. The model architecture addresses two critical constraints: discriminative power and computational efficiency. Our feature engineering process yields the following optimal feature set:

- Cardinality of scanned points
- Distribution of distances among current top-5 candidates
- Temporal distance progression in recent top-5 results
- Relative distance differentials between top-K candidates and optimal solution

To facilitate efficient feature computation, we implement an optimized sorted array structure that replaces conventional heap-based candidate management.

4.4 Labeling-based Automatic ILPs Tuner

Impact of ILPs. The retrieval efficiency and effectiveness of graph-based indices are fundamentally determined by their structural properties. Modifications to ILPs (e.g., maximum degree m_c [30, 41], pruning rate α_c [30]) induce concurrent alterations in both graph topology and search dynamics, creating non-linear interactions between retrieval speed and accuracy. The following example demonstrates these interdependent effects:

Example 6. *Reducing m_c to 1 causes irreparably damaged graph connectivity. Despite asymptotically increasing the search frontier parameter ef_s, this configuration achieves near-zero recall accuracy due to path discontinuity. In contrast, when elevating both m_c and α to extreme values, the graph degenerates into brute-force search* *patterns, with each traversal step requiring a full dataset scan, thereby minimizing retrieval throughput.*

The necessity for index reconstruction during parameter space exploration creates prohibitive computational costs (typically 2-3 hours per rebuild for million-scale datasets), particularly when optimizing multiple ILPs simultaneously.

4.4.1 Compression-Based Construction Framework. We empirically observe that indices constructed with varying ILPs exhibit substantial edge overlap. As formalized in Theorem 4.1, when maintaining a fixed pruning rate α, a graph built with lower maximum degree m_c constitutes a strict subgraph of its higher-degree counterpart.

Theorem 4.1. *(Maximum Degree-Induced Subgraph Hierarchy) Let G^a and G^b denote indices constructed with maximum degrees $m_c = a$ and $m_c = b$ respectively, under identical initialization conditions and fixed α. Given equivalent greedy search outcomes $\text{ANN}_k(x)$ for all points x under both configurations, then $a < b \implies G^a \subset G^b$ where $\forall(x_i, x_j) \in E(G^a), (x_i, x_j) \in E(G^b)$.*

Proof. Please refer to Appendix B of our report [52]. □

VSAG employs an *index compression strategy* that constructs the index once using relaxed ILPs while labeling edges with specific ILPs. During the search process, the algorithm dynamically selects edges by leveraging labels, shifting the edge selection process from the index construction phase to the search phase. It enables adaptive edge selection without requiring index reconstruction. Thus, it achieves equivalent search performance to maintaining multiple indices with different parameter configurations, while incurring only the overhead of constructing a single index. The parameter labeling mechanism maintains topological flexibility while enhancing storage efficiency through a compressed index representation. Due to space limitations, please refer to Appendix B of our report [52] for the detailed construction algorithm of *VSAG*. Then, we illustrate the labeling algorithm, which assigns label to each edge.

Prune-based Labeling Algorithm. The pruning strategy of *VSAG* is shown in Algorithm 2. First, $\text{ANN}_k(x_i)$ and their distances T_i are sorted in ascending order of distance, and the pruning rates A are also sorted in ascending order. The reverse insertion position r indicates whether this pruning process occurs during the insertion of a reverse edge. We initialize the out-edges of x_i by $ANN_k(x_i)_{r:}$ (Line 1). Each edge is then assigned a label of 0 (Line 2). Note that when $r > 0$, it indicates that the current pruning occurs during the reverse edge addition phase, and only the labels of edges within the interval $[r : |G_i|]$ need to be updated. Otherwise, all labels should be updated. We use *count* to record the number of neighbors that have non-zero labels (Line 3). When count $= r$, it means we have already collected all the neighbors we need. At this point, the algorithm should terminate (Lines 4-5).

Next, each α_c is examined in ascending order (Line 4). For each unlabeled neighbor x_j (Lines 5-7), neighbor x_k with smaller distance is used to make pruning decision (Lines 8-9). The pruning decision requires satisfying two conditions (Lines 10-12): (a) The neighbor x_k exists in the graph constructed with the α_c (i.e., $0 < L_{i,k} \le \alpha_c$). (b) The pruning condition is satisfied (i.e., $\alpha_c \cdot \tau(x_j, x_k) \le \tau(x_i, x_j)$). Here, we accelerate the computation of $\tau(x_i, x_j)$ by using the cached result $T_{i,j}$. If no neighbor can prune x_j with α_c, it is assigned the

Algorithm 2: Prune-based Labeling

```
Input: base points $x_i$, approximate nearest neighbors
       $\text{ANN}_k(x_i)$ and related distances $T_i$ sorted by
       distance in ascending order, pruning rates $A$ sorted
       in ascending order, maximum degree $m_c$, reverse
       insertion position $r$
Output: neighbors list $G_i$ and labels list $L_i$ of point $x_i$
1  initialize neighbors list $G_{i,r:} \leftarrow \text{ANN}_k(x_i)_{r:}$
2  initialize neighbors labels list $L_{i,r:}$ with 0
3  $count \leftarrow r$
4  foreach $\alpha_c \in A$ and $count < m_c$ do
5      foreach $j \in G_{i,r:}$ and $count < m_c$ do
6          if $L_{i,j} \neq 0$ then
7              continue
8          is_pruned ← False
9          foreach $x_k \in G_{i,0:j}$ do
10             if $0 < L_{i,k} \le \alpha_c$ and $\alpha_c \cdot \tau(x_j, x_k) \le T_{i,j}$ then
11                 is_pruned ← True
12                 break
13         if not is_pruned then
14             $L_{i,j} \leftarrow \alpha_c$
15             $count \leftarrow count + 1$
16 shrink $|G_i| \le m_c$ and $|L_i| \le m_c$ by removing $x_j$ $s.t.$ $L_{i,j} = 0$
17 return $G_i, L_i$
```

label $L_{i,j} \leftarrow \alpha_c$ (Lines 13-15). Finally, the algorithm returns the neighbor set G_i and labels set L_i of x_i (Lines 16-17).

Building upon the parameter analysis of m_c, we formally establish the subgraph inclusion property for graphs constructed with varying pruning rates α_c in Algorithm 2 through Theorem 4.2.

Theorem 4.2. *(Subgraph Inclusion Property with Varying Pruning Rate α_c) Fix all ILPs except α_c, and let G^a and G^b be indices constructed by Algorithm 3 in our report [52] using pruning rates $\alpha_c = a$ and $\alpha_c = b$ respectively, where $a < b$. Suppose that for every data point x_i, the finite-sized approximate nearest neighbor (ANN) sets $\text{ANN}_k(x_i)$ retrieved during construction remain identical under both α_c values. Then G^a forms a subgraph of G^b, i.e., all edges in G^a satisfy $(x_i, x_j) \in E(G^b)$.*

Proof. Please refer to Appendix C.2 of our report [52]. □

This theorem reveals a monotonic relationship between pruning rates and graph connectivity. When reducing α_c, any edge pruned under this stricter parameter setting would necessarily be eliminated under larger α_c values. This observation enables us to characterize each edge by its preservation threshold α_e: the minimal pruning rate required to retain the edge during construction. Consequently, all graph indices constructed with pruning rates $\alpha_c \ge \alpha_e$ will contain this edge. This threshold-based perspective permits efficient compression of multiple parameterized graph structures into a unified index, where edges are annotated with their respective α_e values.

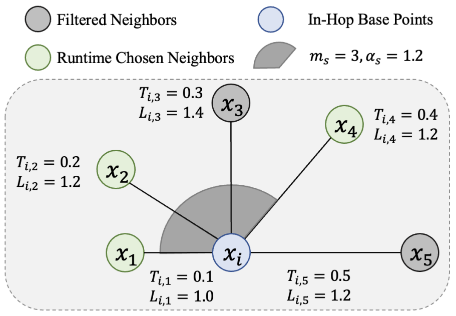


Figure 5: Runtime Adjust ILP(m_c and α_c) by Tuning m_s and α_s

Example 7. *As shown in Figure 5, we illustrate the state of the labeled graph generated by Algorithm 2 and the runtime edge selection process. Suppose that during the greedy search in the graph using Algorithm 1, we need to explore the in-hop base point x_i (Line 5 of Algorithm 1). The node x_i has 5 neighbors sorted by distance $T_{i,j}$ in ascending order (i.e., $x_1, \ldots, x_5$). The distances are $T_{i,1}, \ldots, T_{i,5}$, and the corresponding labels are $L_{i,1}, \ldots, L_{i,5}$.*

Given a relaxed ILP with $m_s = 3$ and $\alpha_s = 1.2$, we visit the neighbors of x_i in ascending order of distance. We then filter out neighbors that do not satisfy the pruning condition (Line 8 of Algorithm 1):

- *x_2 is filtered out because $L_{i,2} = 1.4 > \alpha_s$.*
- *x_5 is filtered out because we have found $m_s = 3$ valid neighbors.*

Thus, in this search hop, we will visit x_1, x_3, and x_4. As proven in Theorem 4.2 and Theorem 4.1, the search process is equivalent to searching in a graph constructed with $m_c = 3$ and $\alpha_c = 1.2$. In other words, we can dynamically adjust the ILPs (i.e., m_c, α_c) by tuning the relaxed QLPs (i.e., m_s, α_s). This approach saves significant costs associated with rebuilding the graph.

Please refer to Appendix D of our report [52] for details of tuning ILPs after VSAG is constructed with labels.

5 DISTANCE COMPUTATION ACCELERATION

Recent studies [21, 53, 54] illustrate that the exact distance computation takes the majority of the time cost of graph-based ANNS. Approximate distance techniques, such as scalar quantization, can accelerate this process at the cost of reduced search accuracy. *VSAG* adopt a two-stage approach that first performs an approximate distance search followed by exact distance re-ranking. §5.1 analyzes the distance computation scheme, with subsequent sections detailing optimization strategies for *VSAG* component.

5.1 Distance Computation Cost Analysis

VSAG employs low-precision vectors during graph traversal operations while reserving precise distance computations exclusively for final result reranking. The dual-precision architecture effectively minimizes distance computation operations (DCO) [53] overhead while preserving search accuracy through precision-aware hierarchical processing.

If we only consider the cost incurred by distance computation, the total distance computation cost can be expressed as follows:

$$cost = cost_{lp} + cost_{hp} = n_{lp} \cdot t_{lp} + n_{hp} \cdot t_{hp} \quad (1)$$

Here, distance computation cost *cost* consists of two components: the computation cost for low-precision vectors $cost_{lp}$ and the computation cost for high-precision vectors $cost_{hp}$. Each component is determined by the number of distance computations (n_{lp} or n_{hp}) and the cost of a single distance computation (t_{lp} or t_{hp}).

The optimization of n_{lp} is closely related to the specific algorithm workflow, while t_{hp} is primarily determined by the computational cost of FLOAT32 vector operations - both of which remain relatively constant. Consequently, the *VSAG* framework focuses primarily on optimizing the parameters t_{lp} and n_{hp}.

VSAG optimizes the overall cost in three ways:

- The combination of quantization techniques, hardware instruction set SIMD, and memory-efficient storage (§5.2) achieves exponential reduction in low-precision distance computation (t_{lp}).
- Enhanced quantization precision by parameter optimization (§5.3) mitigates candidate inflation from precision loss, achieving sublinear growth in required low-precision computations (n_{hp}).
- Selective re-ranking with dynamic thresholding (§5.4) establishes an accuracy-efficiency equilibrium, restricting high-precision validation (n_{hp}) to a logarithmically scaled candidate subset.

5.2 Minimizing Low-Precision Computation Overhead

SIMD and Quantization Methods. Modern CPUs employ SIMD instruction sets (SSE/AVX/AVX512) to accelerate distance computations through vectorized operations. These instructions process 128-bit, 256-bit, or 512-bit data chunks in parallel, with vector compression techniques enabling simultaneous processing of multiple vectors. For example, AVX512 can compute one distance for 16-dimensional FLOAT32 vectors per instruction, but when compressing vectors to 128 bits, it achieves 4x acceleration by processing four vector pairs concurrently. Product Quantization (PQ) [31] enables high compression ratios for batch processing through SIMD-loaded lookup tables. While PQ-Fast Scan [4] excels in partition-based searches through block-wise computation, its effectiveness diminishes in graph-based searches due to random vector storage patterns and inability to filter visited nodes, resulting in wasted SIMD bandwidth. In contrast, Scalar Quantization (SQ) [59] proves more suitable for graph algorithms by directly compressing vector dimensions (e.g., FLOAT32$\rightarrow$ INT8/INT4) without requiring lookup tables. As demonstrated in *VSAG*, SQ achieves the optimal balance between compression ratio and precision preservation while fully utilizing SIMD acceleration capabilities, making it particularly effective for memory-bound graph traversals.

Distance Decomposition. *VSAG* optimizes Euclidean distance computations by decoupling static and dynamic components. The system precomputes and caches invariant vector norms during database indexing, then combines them with real-time dot product computations during queries. This decomposition reduces operational complexity while preserving mathematical equivalence, as shown by the reformulated Euclidean distance:

$$||x_b - x_q||^2 = ||x_b||^2 + ||x_q||^2 - 2x_b \cdot x_q,$$

The computational optimization strategy can be summarized as follows: Only the Inner Product term $x_b \cdot x_q$ requires real-time computation during search operations, while the squared query norm $||x_q||^2$ can be pre-computed offline before initiating the search process. By storing just one additional FLOAT32 value per database vector x_b (specifically the precomputed $||x_b||^2$), we can effectively

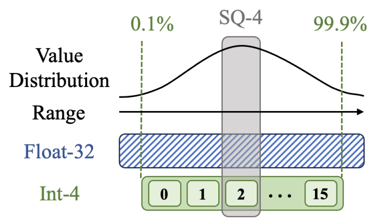


Figure 6: Truncated scalar quantization.

transform the computationally expensive Euclidean distance computation into an equivalent inner product operation. This space-time tradeoff reduces the subtraction CPU instruction in distance computation, which saves one CPU clock cycle.

5.3 Improving Quantization Precision

Scalar quantization (SQ) compresses floating-point vectors by mapping 32-bit float values to lower-bit integer representations in each dimension. In SQ-b (where b denotes bit-width), the dynamic range is uniformly partitioned into 2^b intervals. For SQ4 (4-bit), this creates 16 intervals over $[0, 1]$, where the k-th interval corresponds to $[(k-1)/15, k/15)$. Values within each interval are encoded as their corresponding integer index. However, practical implementations face critical range estimation challenges: direct use of observed min/max values proves suboptimal when data outliers existing.

Example 8. *Analysis of the GIST1M dataset reveals this limitation – while 99% of dimensions exhibit values below 0.3, using the absolute maximum (1.0) would leave 70% of the quantization intervals (0.3-1.0) underutilized. This interval under-utilization severely degrades quantization precision.*

To enhance robustness, we propose *Truncated Scalar Quantization* using the 99th percentile statistics rather than absolute extremes. This approach discards outlier-induced distortions while preserving quantization resolution over the primary data distribution, achieving superior balance between compression efficiency and numerical precision as shown in Figure 6.

5.4 Selective Re-rank

Quantization methods can significantly enhance retrieval efficiency, but quantization errors may lead to substantial recall rate degradation. While re-ranking with full-precision vectors can mitigate this performance loss. However, applying exhaustive re-ranking to all candidates is inefficient. The *VSAG* framework addresses this challenge through selective re-ranking, effectively compensating for approximation errors in distance computation without compromising system performance. A straightforward approach is to select only the candidates with small low-precision distances for re-ranking. The optimal number of candidates requiring re-ranking varies significantly depending on query characteristics, quantization error distribution, and search requirement k. To address this dynamic requirement, *VSAG* implements DDC [53] scheme that can automatically adapt re-ranking scope based on error-distance correlation analysis.

Table 3: Dataset Statistics.

Dataset	Dim	#Base	#Query	Type
GIST1M	960	1,000,000	1,000	Image
SIFT1M	128	1,000,000	1,000	Image
TINY	384	5,000,000	1,000	Image
GLOVE-100	100	1,183,514	10,000	Text
WORD2VEC	300	1,000,000	1,000	Text
OPENAI	1536	999,000	1,000	Text

Table 4: Parameter Settings of Algorithms.

Algorithm	Parameter Settings
VSAG	`maximum_degree` $\in \{8, 12, 16, 24, 32, 36, 48, 64\}$ `pruning_rate` $\in \{1.0, 1.2, 1.4, 1.6, 1.8, 2.0\}$
hnswlib, hnsw(faiss)	`maximum_degree` $\in \{4, 8, 12, 16, 24, 36, 48, 64, 96\}$
nndescent	`pruning_prob` $\in [0.0, 1.0]$ `leaf_size` $\in \{24, 36, 48\}$ `n_neighbors` $\in \{10, 20, 40, 60\}$ `pruning_degree_multiplier` $\in \{0.5, 0.75, 1.0, 1.5, 2.0, 3.0\}$
faiss-ivf, faiss-ivfpqfs	`n_clusters` $\in \{32, 64, 128, 256, 512, 1024, 2048, 4096, 8192\}$
scann	`n_leaves` $\in \{100, 600, 1000, 1500, 2000\}$ `avq_threshold` $\in \{0.15, 0.2, 0.55\}$ `dims_per_block` $\in \{1, 2, 3, 4\}$

6 EXPERIMENTAL STUDY

6.1 Experimental Setting

Datasets. Table 3 presents the datasets used in our experiments, and they are widely adopted in existing works [5] and benchmarks [20]. For each dataset, we report the vector dimensions (Dim), the number of base vectors (#Base), the number of query vectors (#Query), and the dataset type (Type). All vectors are stored in `float32` format.

Algorithms. We compare *VSAG* with three graph-based methods (*hnswlib, hnsw(faiss), nndescent*) and three partition-based methods (*faiss-ivf, faiss-ivfpqfs, scann*). All methods are widely adopted in industry, and Faiss is the most popular vector search library.

- *hnswlib* [39]: the most popular graph-based index.
- *hnsw(faiss)* [34]: the HNSW implementation in Faiss.
- *nndescent* [14]: a graph-based index that achieves efficient index construction by iteratively merging the neighbors of base vectors.
- *faiss-ivf* [34]: the most popular partition-based method.
- *faiss-ivfpqfs.* [4]: an IVF implementation with PQ (Product Quantization) [32] and FastScan [4] optimizations.
- *scann.* [26]: The partition-based method developed by Google, which is highly optimized for Maximum Inner Product Search (MIPS) through anisotropic vector quantization.

Performance Metrics. We evaluate algorithms with *Recall Rate* and *Queries Per Second (QPS)* [30, 41]. The recall rate is defined as the percentage of actual KNNs among the vectors retrieved by the algorithm, i.e., Recall@k = $\frac{|\mathrm{ANN}_k(x_q) \cap \mathrm{NN}_k(x_q)|}{k}$. If not specified, the QPS is the queries per second when $k = 10$. Each algorithm is evaluated on a dedicated single core.

Parameters. Table 4 reports the parameter configurations of algorithms. Unless otherwise specified, we report the best performance of an algorithm among all combinations of parameter settings.

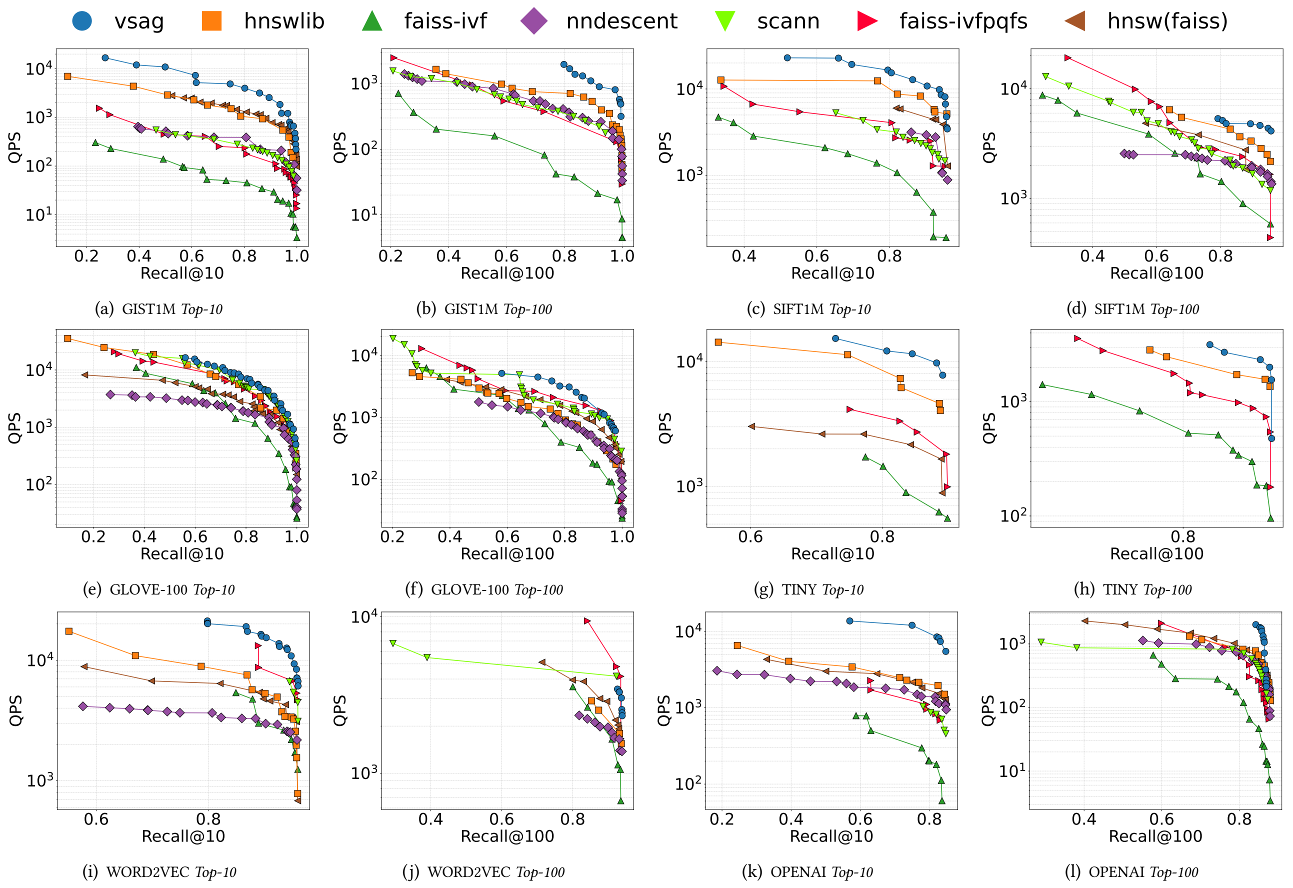


Figure 7: Overall Performance.

Environment. The experiments are conducted on a server with an Intel(R) Xeon(R) Platinum 8163 CPU @ 2.50GHz and 512GB memory, except that Section 6.2 is evaluated on an AWS `r6i.16xlarge` machine with hyperthreading disabled. We implement *VSAG* in C++, and compile it with g++ 10.2.1, `-Ofast` flag, and AVX-512 instructions enabled. For other baselines, we use the implementations from the official Docker images.

6.2 Overall Performance

Figure 7 evaluates the recall (Recall@10 and Recall@100) vs. QPS performance of algorithms. We report the best performance of an algorithm under all possible parameter settings. Across all datasets, *VSAG* can achieve higher QPS with the same recall rate. In addition, *VSAG* can provide a higher QPS increase on high-dimensional vector datasets, such as GIST1M and OPENAI. In particular, *VSAG* outperforms *hnswlib* by 226% in QPS on GIST1M when we ensure *Recall*@10 = 90%, it also provides 400% higher QPS than *hnswlib* on OPENAI when ensuring *Recall*@10 = 80%. The reason is that *VSAG* adopts quantization methods, which can provide significant QPS increase without sacrificing the search accuracy, and they are especially effective for high-dimensional data [22].

6.3 Ablation Study

6.3.1 Cache Miss Analysis. Table 5 conduct ablation tests to investigate the effectiveness of *VSAG*'s strategies (see §3,§4 and§5). We use GIST1M and SIFT1M datasets, and set $m_c = 36, \alpha_c = 1.0$ for *VSAG*. The strategies are incremental, i.e., the strategy k row reports the performance of the baseline with strategies 1..k.

Strategy 1 uses *quantization methods*, and it improves QPS while ensuring the same recall rates. Specifically, the QPS on GIST1M increases from 510 to 1272 (149% growth), and the QPS on SIFT1M increases from 1695 to 2881 (69% growth). This is because quantization can significantly reduce the cost of distance computation.

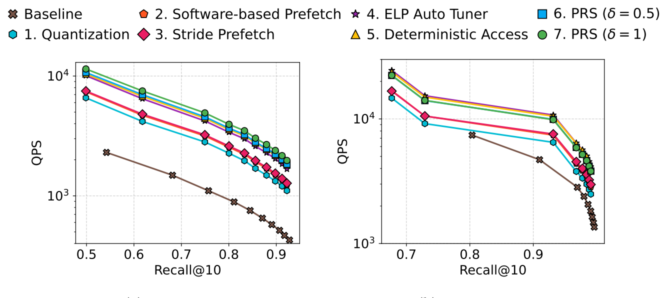
Strategies 2-5 optimize *memory access*. As a result, the L3 cache miss rate reduces from 93.89% to 39.23% on GIST1M, and from 77.88% to 20.98% on SIFT1M. Such a drop in cache miss rate leads to a 70% QPS increase on GIST1M , and a 74% QPS increase on SIFT1M. This is because L3 cache loads are the dominate cost in the search time after using quantization methods.

Strategies 6-7 use *PRS* to balance memory and CPU usage. When δ increase, *VSAG* would allocates more memory, and the L3 cache load of will decrease. For example, the L3 cache load decrease by 41% on GIST1M, and by 30% on SIFT1M. On GIST1M dataset, *VSAG*'s QPS increases from 2167 to 2337, as memory pressure is the performance bottleneck. However, on SIFT1M dataset, *VSAG*'s QPS on GIST1M decreases from 5027 to 4640, because CPU pressure outweighs memory pressure on this dataset. We can decide whether to use *PRS* based on the workload of memory and CPU.

6.3.2 Performance. Figure 8 illustrates the cumulative performance gaps under varying numbers of optimization strategies. For each

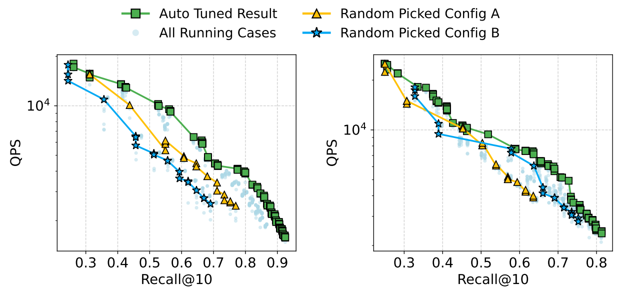
Table 5: Ablation Study of *VSAG*'s Strategies.

Strategy	Recall@10		QPS		L3 Cache Load		L3 Cache Miss Rate		L1 Cache Miss Rate	
	GIST1M	SIFT1M	GIST1M	SIFT1M	GIST1M	SIFT1M	GIST1M	SIFT1M	GIST1M	SIFT1M
Baseline	90.7%	99.7%	510	1695	198M	112M	93.89%	77.88%	39.37%	17.55%
Above + 1.Quantization	89.8%	98.4%	1272	2881	125M	79M	67.42%	52.09%	19.44%	11.56%
Above + 2. Software-based Prefetch	89.8%	98.4%	1490	3332	120M	53M	71.71%	53.86%	16.98%	9.58%
Above + 3. Stride Prefetch	89.8%	98.4%	1517	3565	118M	50M	64.57%	19.26%	17.18%	9.66%
Above + 4. ELP Auto Tuner	89.8%	98.4%	2052	4946	43M	49M	45.88%	32.65%	16.44%	10.11%
Above + 5. Deterministic Access	89.8%	98.4%	2167	5027	65M	72M	39.23%	20.98%	15.43%	9.91%
Above + 6. PRS ($\delta = 0.5$)	89.8%	98.4%	2255	4668	55M	63M	55.75%	50.74%	15.20%	10.17%
Above + 7. PRS ($\delta = 1$)	89.8%	98.4%	2377	4640	46M	55M	71.62%	74.73%	14.69%	9.26%



(a) GIST1M (b) GLOVE-100

Figure 8: Performance of *VSAG*'s Strategies.



(a) GIST1M (b) GLOVE-100

Figure 9: Performance of Auto-Tuned ILPs.

strategy, varying ef_s from 10 to 100 yields specific Recall@10 and QPS values. For example, the top-left point in the Baseline corresponds to $ef_s = 10$, while the bottom-right point corresponds to $ef_s = 100$. The strategies contributing most to QPS improvements are 1. Quantization and 4. ELP Auto Tuner. The former drastically reduces distance computation overhead, while the latter significantly enhances the prefetch effectiveness in 3. Stride Prefetch. However, applying only 3. Stride Prefetch shows minimal difference compared to 2. Software-based Prefetch, as prefetch efficiency heavily depends on environment-level parameters ω and ν.

6.4 Evaluation of ILP Auto-Tuner

6.4.1 Tuning Cost. We evaluate the tuning time and memory footprint when varying index-level parameters $m_c \in (8, 16, 24, 32)$ and $\alpha_c \in (1.0, 1.2, 1.4, 1.6, 1.8, 2.0)$. The compared method including *FIX* (fix parameter $m_c = 32$ and $\alpha_c = 2.0$, building with *hnswlib*) and *BF* (construct indexes of all combination of index-level parameters, building with *hnswlib*). Table 6 shows results (*Mem* for memory footprint in gigabytes; *Time* for building time in hours).

Among the four datasets, the method with minimal construction time and memory usage is *FIX*, while *BF* requires the most. This is because baseline methods can only brute-force traverse each parameter configuration and build indexes separately. Additionally, it can dynamically select edges during search using edge labels acquired during construction, enabling parameter adjustment at search time without repeated index construction. This allows *VSAG* to achieve a 20x tuning time saving compared to *BF* on GIST1M. Moreover, the use of T to cache distances during construction eliminates redundant computations, further accelerating *VSAG*'s tuning. In terms of memory usage, graphs built with different m_c and a_c share significant structural overlap. This enables *VSAG* to compress these graphs via edge labels, resulting in over 20x memory footprint reduction during tuning compared to *BF*. Compared to *FIX*, *VSAG* only introduces minimal additional memory costs for storing extra edges and label sets.

6.4.2 Tuning Performance. Beyond tuning costs, we also demonstrate the performance of the ILPs auto tuned index. As shown in Figure 9, we randomly selected two index-level parameter configurations (A and B) as baselines and plotted the performance of all parameter combinations in running cases. *VSAG* exhibits significant performance gains over these baselines. For example, on the GIST1M dataset at a fixed QPS of 2500, the worst-case Recall@10 among all running cases is 62%, while the tuned index achieves Recall@10=88%, representing a 26% (absolute) improvement. At a fixed Recall@10=70%, the worst-case QPS is 2000, while *VSAG* achieves 4000 QPS - an improvement of 100%. Similar trends hold for the GLOVE-100 dataset: maximum Recall@10 improvements exceed 15% at fixed QPS of 500, and QPS improves from around 4000 to 7000 (over 75% gain) at a fixed recall rate of 60%.

6.5 Evaluation of QLP Auto-Tuner

Table 7 evaluates the QLPs tuning result on GIST1M and SIFT1M, under a recall guarantee of 94% and 97%. The baseline method (i.e., *FIX*) manually selects the smallest ef_s that ensures the target recall. In contrast, *VSAG* employs a decision tree classification approach to divide queries into two categories: (1) Simple queries, which can converge to the target accuracy with a smaller ef_s value and incur lower computational cost. For these queries, an appropriate ef_s can help improve retrieval speed. (2) Complex queries, which require larger ef_s values to achieve the desired accuracy. For these queries, an appropriate ef_s can help improve retrieval precision. For both types of queries, the QLP Auto Tuner can intelligently

Table 6: Comparison of Tuning Cost of ILPs.

Dataset	FIX		BF		*VSAG* (Ours)	
	Mem	Time	Mem	Time	Mem	Time
GIST1M	3.83G	3.20H	89.87G	61.64H	4.07G	2.92H
SIFT1M	0.73G	0.85H	15.48G	30.79H	0.97G	1.86H
GLOVE-100	0.74G	1.22H	15.36G	41.94H	1.03G	2.13H

selects the required ef_s, leading to over a 5% increase in QPS when recall thresholds equal 94% and 97%, respectively.

7 CASE STUDIES AND APPLICATIONS

Product Search and Recommendation Systems. Vector retrieval is widely used in product search and recommendation systems within the e-commerce industry. By converting information, such as user behavior, browsing history, and purchase history, into vectors, the vector similarity search can identify products that align with user preferences, thereby enabling personalized search and recommendation. E-commerce platforms [3, 47] often handle millions of vectors of product and user, necessitating efficient storage and retrieval capabilities. These applications are latency-sensitive, placing high demands on the efficiency of retrieval algorithms. Taking advantage of memory access optimization and distance computation optimization in *VSAG*, search and recommendation services can achieve a higher QPS per unit of computational resource (per core). In an image search scenario using 512-dimensional vectors at Ant Group, *VSAG* reduces service resource consumption by 4.6x and decreases service latency by 1.2x compared to the latest partition-based methods in industry.

In production environments, there are instances with different *resource classes* (e.g., 4C16G and 2C8G) and *platforms* (e.g., x86, ARM). However, achieving the best performance across these different resource classes and platforms requires extensive manual parameters tuning. Moreover, the tuning results are difficult to transfer, and changing resource classes often necessitates re-tuning. The automatic parameter tuning tool provided by *VSAG* significantly reduces the deployment and tuning costs of vector similarity search services, enabling efficient retrieval across different platforms, environments, and retrieval requirements.

Large Language Model. To reduce hallucinations and provide *real-time responses*, Large Language Model (LLM) [36] conversations typically require relevant text content as contextual input. Text vectors generally have higher dimensions [23]. For example, OpenAI's latest text-embedding-3-small[2] outputs a 1,536-dimensional vector, while text-embedding-3-large outputs a 3,072-dimensional vector. In contrast, the commonly used vector dimension for images is 256. It requires algorithms capable of efficiently and accurately storing and retrieving large amounts of high-dimensional vectors. The distance computation optimization and memory access optimization make *VSAG* particularly suitable for efficiently retrieving such vectors.

Recently, DeepSearch (e.g., DeepSeek-R1 [13]) is gradually emerging as the new standard for LLM, with its underlying chain-of-thought reasoning [45] heavily reliant on relevant textual content input. Multi-step reasoning [33, 57] of Retrieval-Augmented Generation (RAG) [23] imply *multi-rounds* of high-dimensional vectors retrieval, posing greater challenges to the throughput and latency of vector retrieval services. For example, in the RAG service using 1,536-dimensional vectors, *VSAG* can provide up to 5x information retrieval capability for LLMs. This enables the model to incorporate more domain-specific information within the same latency.

[2]https://platform.openai.com/docs/guides/embeddings

Table 7: Comparison of Tuning Performance of QLP

Method	Metric	GIST1M		SIFT1M	
		94%	97%	94%	97%
FIX	Recall@10	94.64%	97.49%	94.54%	97.61%
	QPS	1469	902	4027	2834
***VSAG* (Ours)**	Recall@10	94.71%	97.58%	94.63%	97.66%
	QPS	1534	967	4050	2912

8 RELATED WORK

The mainstream methods for vector retrieval can be divided into two categories: space partitioning-based and graph-based. Graph-based algorithms [7, 16, 17, 29, 37, 38, 40, 42] can ensure high recall with practical efficiency, e.g., HNSW [40], NSG [18], VAMANA [29], and τ-MNG [43]. These methods build a proximity graph where each node is a base vector and edges connect pairs of nearby vectors. During a vector search, they greedily move towards the query vector to identify its nearest neighbors. Our *VSAG* framework can adapt to graph-based algorithms mentioned above, to improve the performance in production.

Space partitioning-based methods (e.g., IVFADC [4, 8, 15, 51]) group similar vectors into subspaces with K-means [4, 35] or Locality-Sensitive Hashing (LSH) [12, 19, 24, 44, 46, 58]. During the search process, they traverse some vector subspaces to find the nearest neighbors. These methods can achieve high cache hit rates due to the continuous organization of vectors, but they suffer from a low recall issue. In comparison, graph-based ANNS algorithms (e.g., *VSAG* and HNSW) usually achieve a higher QPS under the same recall. Note that the technique of *VSAG* cannot be applied to space partitioning-based algorithms.

9 CONCLUSION

In this work, we present *VSAG*, an optimized search framework for ANNS that can be applied to any graph-based index. To address the challenge of high memory access costs, we employ software-based prefetch, deterministic access greedy search, and PRS to significantly reduce the cache miss rate. For the challenge of high parameter tuning costs, we propose a three-level parameter tuning mechanism that automatically adjusts different parameters based on their tuning complexity. To tackle the issue of high distance computation costs, we combine quantization and selective re-ranking to integrate low- and high-precision distance computations, thereby enhancing search efficiency without sacrificing effectiveness. Experiments on real-world datasets demonstrate that *VSAG* outperforms baselines in both retrieval performance and parameter tuning cost.

REFERENCES

[1] Alipay. 2025. Ant Group. https://www.antgroup.com.
[2] Alipay. 2025. Face Recognition. https://open.alipay.com/api/detail?code=I1080300001000043632.
[3] Amazon. 2025. Product Search. https://www.amazon.com/.

[4] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2015. Cache locality is not enough: High-Performance Nearest Neighbor Search with Product Quantization Fast Scan. *Proceedings of the VLDB Endowment* 9, 4 (2015).

[5] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems* 87 (2020), 101374.

[6] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. 2020. Classifying Memory Access Patterns for Prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 513–526. https://doi.org/10.1145/3373376.3378498

[7] Ilias Azizi, Karima Echihabi, and Themis Palpanas. 2023. Elpis: Graph-based similarity search for scalable data science. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1548–1559.

[8] Artem Babenko and Victor S. Lempitsky. 2015. The Inverted Multi-Index. *IEEE Trans. Pattern Anal. Mach. Intell.* 37, 6 (2015), 1247–1260.

[9] Peter Braun and Heiner Litz. 2019. Understanding memory access patterns for prefetching. In *International Workshop on AI-assisted Design for Architecture (AIDArc), held in conjunction with ISCA*.

[10] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi (Eds.). ACM, 785–794. https://doi.org/10.1145/2939672.2939785

[11] Tien-Fu Chen and Jean-Loup Baer. 1995. Effective hardware-based data prefetching for high-performance processors. *IEEE transactions on computers* 44, 5 (1995), 609–623.

[12] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*. 253–262.

[13] DeepSeek. 2025. DeepSeek. https://www.deepseek.com/.

[14] Wei Dong, Charikar Moses, and Kai Li. 2011. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World wide web*. 577–586.

[15] Yihe Dong, Piotr Indyk, Ilya P Razenshteyn, and Tal Wagner. 2020. Learning Space Partitions for Nearest Neighbor Search. *ICLR* (2020).

[16] Cong Fu, Changxu Wang, and Deng Cai. 2022. High Dimensional Similarity Search With Satellite System Graph: Efficiency, Scalability, and Unindexed Query Compatibility. *IEEE Trans. Pattern Anal. Mach. Intell.* 44, 8 (2022), 4139–4150.

[17] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *Proc. VLDB Endow.* 12, 5 (2019), 461–474.

[18] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proc. VLDB Endow.* 12, 5 (Jan. 2019), 461–474. https://doi.org/10.14778/3303753.3303754

[19] Junhao Gan, Jianlin Feng, Qiong Fang, and Wilfred Ng. 2012. Locality-sensitive hashing scheme based on dynamic collision counting. In *Proceedings of the 2012 ACM SIGMOD international conference on management of data*. 541–552.

[20] Jianyang Gao, Yutong Gou, Yuexuan Xu, Yongyi Yang, Cheng Long, and Raymond Chi-Wing Wong. 2024. Practical and Asymptotically Optimal Quantization of High-Dimensional Vectors in Euclidean Space for Approximate Nearest Neighbor Search. *arXiv preprint arXiv:2409.09913* (2024).

[21] Jianyang Gao and Cheng Long. 2023. High-Dimensional Approximate Nearest Neighbor Search: with Reliable and Efficient Distance Comparison Operations. *Proc. ACM Manag. Data* 1, 2 (2023), 137:1–137:27. https://doi.org/10.1145/3589282

[22] Jianyang Gao and Cheng Long. 2024. RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 2, 3 (2024), 167. https://doi.org/10.1145/3654970

[23] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. 2024. Retrieval-Augmented Generation for Large Language Models: A Survey. arXiv:2312.10997 [cs.CL] https://arxiv.org/abs/2312.10997

[24] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. 1999. Similarity search in high dimensions via hashing. In *Vldb*, Vol. 99. 518–529.

[25] Google. 2025. Search Engine. https://www.google.com/.

[26] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*. PMLR, 3887–3896.

[27] 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.

[28] Intel. 2025. AVX512. https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/what-is-intel-avx-512.html.

[29] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in Neural Information Processing Systems* 32 (2019).

[30] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in neural information processing Systems* 32 (2019).

[31] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Trans. Pattern Anal. Mach. Intell.* 33, 1 (2011), 117–128. https://doi.org/10.1109/TPAMI.2010.57

[32] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Trans. Pattern Anal. Mach. Intell.* 33, 1 (2011), 117–128.

[33] Kaixuan Ji, Guanlin Liu, Ning Dai, Qingping Yang, Renjie Zheng, Zheng Wu, Chen Dun, Quanquan Gu, and Lin Yan. 2025. Enhancing Multi-Step Reasoning Abilities of Language Models through Direct Q-Function Optimization. arXiv:2410.09302 [cs.LG] https://arxiv.org/abs/2410.09302

[34] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.

[35] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.

[36] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[37] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2020. Approximate Nearest Neighbor Search on High Dimensional Data - Experiments, Analyses, and Improvement. *IEEE Trans. Knowl. Data Eng.* 32, 8 (2020), 1475–1488.

[38] Kejing Lu, Mineichi Kudo, Chuan Xiao, and Yoshiharu Ishikawa. 2021. HVS: Hierarchical Graph Structure Based on Voronoi Diagrams for Solving Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 15, 2 (2021), 246–258.

[39] Yu A Malkov and Dmitry A Yashunin. [n.d.]. hnswlib. https://github.com/nmslib/hnswlib.

[40] Yury A. Malkov and Dmitry A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 42, 4 (2020), 824–836.

[41] Yury A. Malkov and Dmitry A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 42, 4 (2020), 824–836. https://doi.org/10.1109/TPAMI.2018.2889473

[42] Yun Peng, Byron Choi, Tsz Nam Chan, Jianye Yang, and Jianliang Xu. 2023. Efficient Approximate Nearest Neighbor Search in Multi-dimensional Databases. *Proc. ACM Manag. Data* 1, 1 (2023), 54:1–54:27. https://doi.org/10.1145/3588908

[43] Yun Peng, Byron Choi, Tsz Nam Chan, Jianye Yang, and Jianliang Xu. 2023. Efficient Approximate Nearest Neighbor Search in Multi-dimensional Databases. *Proc. ACM Manag. Data* 1, 1, Article 54 (May 2023), 27 pages. https://doi.org/10.1145/3588908

[44] Maxim Raginsky and Svetlana Lazebnik. 2009. Locality-sensitive binary codes from shift-invariant kernels. *Advances in neural information processing systems* 22 (2009).

[45] Zhenyi Shen, Hanqi Yan, Linhai Zhang, Zhanghao Hu, Yali Du, and Yulan He. 2025. CODI: Compressing Chain-of-Thought into Continuous Space via Self-Distillation. arXiv:2502.21074 [cs.CL] https://arxiv.org/abs/2502.21074

[46] Yifang Sun, Wei Wang, Jianbin Qin, Ying Zhang, and Xuemin Lin. 2014. SRS: Solving c-Approximate Nearest Neighbor Queries in High Dimensional Euclidean Space with a Tiny Index. *Proc. VLDB Endow.* 8, 1 (2014), 1–12.

[47] Taobao. 2025. Product Search. https://www.taobao.com/.

[48] Steven P Vanderwiel and David J Lilja. 2000. Data prefetch mechanisms. *ACM Computing Surveys (CSUR)* 32, 2 (2000), 174–199.

[49] Alexander V Veidenbaum, Weiyu Tang, Rajesh Gupta, Alexandru Nicolau, and Xiaomei Ji. 1999. Adapting cache line size to application behavior. In *Proceedings of the 13th international conference on Supercomputing*. 145–154.

[50] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. 2021. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*. 2614–2627.

[51] Jingdong Wang, Ting Zhang, Nicu Sebe, Heng Tao Shen, et al. 2017. A survey on learning to hash. *IEEE transactions on pattern analysis and machine intelligence* 40, 4 (2017), 769–790.

[52] Jiabao Jin Mingyu Yang Deming Chu Xiangyu Wang Zhitao Shen Wei Jia George Gu Yi Xie Xuemin Lin Heng Tao Shen Jingkuan Song Peng Cheng Xiaoyao Zhong, Haotian Li. 2025. VSAG: An Optimized Search Framework for Graph-based Approximate Nearest Neighbor Search [technical report]. http://cspcheng.github.io/pdf/VSAG.pdf. (2025).

[53] Mingyu Yang, Wentao Li, Jiabao Jin, Xiaoyao Zhong, Xiangyu Wang, Zhitao Shen, Wei Jia, and Wei Wang. 2024. Effective and General Distance Computation for Approximate Nearest Neighbor Search. *arXiv preprint arXiv:2404.16322* (2024).

[54] Mingyu Yang, Wentao Li, and Wei Wang. 2024. Fast High-dimensional Approximate Nearest Neighbor Search with Efficient Index Time and Space. *arXiv preprint arXiv:2411.06158* (2024).

[55] Tiannuo Yang, Wen Hu, Wangqi Peng, Yusen Li, Jianguo Li, Gang Wang, and Xiaoguang Liu. 2024. VDTuner: Automated Performance Tuning for Vector Data Management Systems. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 4357–4369. https://doi.org/10.1109/ICDE60146.2024.00332

[56] YouTube. 2025. Video Search. https://www.youtube.com/.

[57] Kongcheng Zhang, Qi Yao, Baisheng Lai, Jiaxing Huang, Wenkai Fang, Dacheng Tao, Mingli Song, and Shunyu Liu. 2025. Reasoning with Reinforced Functional Token Tuning. arXiv:2502.13389 [cs.AI] https://arxiv.org/abs/2502.13389

[58] Bolong Zheng, Xi Zhao, Lianggui Weng, Nguyen Quoc Viet Hung, Hang Liu, and Christian S. Jensen. 2020. PM-LSH: A Fast and Accurate LSH Framework for High-Dimensional Approximate NN Search. *Proc. VLDB Endow.* 13, 5 (2020), 643–655.

[59] Wengang Zhou, Yijuan Lu, Houqiang Li, and Qi Tian. 2012. Scalar quantization for large scale image search. In *Proceedings of the 20th ACM Multimedia Conference, MM '12, Nara, Japan, October 29 - November 02, 2012*, Noboru Babaguchi, Kiyoharu Aizawa, John R. Smith, Shin'ichi Satoh, Thomas Plagemann, Xian-Sheng Hua, and Rong Yan (Eds.). ACM, 169–178. https://doi.org/10.1145/2393347.2393377

Algorithm 3: *VSAG* Index Construction

```
Input: dataset $D$, pruning rates $A$ sorted in ascending
       order, candidate size $ef_c$, maximum degree $m_c$,
Output: directed graph $G$ with out-degree $\leq m_c$
1  initialize $G$ to an empty graph
2  initialize $L, T$ to empty lists
3  for $0 \leq i < |D|$ do
4      $x_i \leftarrow D_i$
       // search ANN
5      $\mathrm{ANN}_k(x_i), T_i \leftarrow$
         GreedySearch($G, L, D, \{x_0\}, x_i, ef_c, m_c, \max(A), 1$)
       // prune
6      $G_i, L_i \leftarrow$ PrunebasedLabeling($x_i, \mathrm{ANN}(x_i), T_i, A, m_c, 0$)
7      foreach $j \in G_i$ do
8          if $T_{i,j} \geq T_{j,-1}$ and $|G_j| \geq m_c$ then
9              continue;
10         else
               // insertion maintaining ascending order
11             $r \leftarrow$ minimum id s.t. $T_{j,r} > T_{i,j}$
12             Insert $x_i$ right before $G_{j,r}$
13             Insert $T_{i,j}$ right before $T_{j,r}$
               // add reverse edges and prune
14             $G_j, L_j \leftarrow$
                 PrunebasedLabeling($x_j, G_{j,r}, T_j, A, m_c, r$)
15 return $G, L$
```

A DETAIL EXPLANATION OF DETERMINISTIC ACCESS GREEDY SEARCH

The initialization phase establishes candidate and visited sets (Lines 1-3). The candidate set dynamically maintains the nearest points discovered during traversal, while the visited set tracks explored points. The algorithm iteratively extracts the nearest unvisited point from the candidate set and processes its neighbors through four key phases (Lines 4-17).

At each iteration, the nearest unvisited point x_i is retrieved from the candidate set (Line 5). Batch processing then identifies *valid* neighbors through three filtering criteria (Lines 6-10): 1) unvisited ($j \notin V$), 2) label validity ($L_j \leq \alpha_s$), and 3) degree constraint ($|N| \leq m_s$). The latter two conditions implement automatic index-level parameters tuning as detailed in Section 4. It is important to note that we differentiate between the identifier j and vector data x_j - a distinction that impacts cache performance as memory access may incur cache misses while identifier operations do not.

The prefetching mechanism operates through two parameters: ω controls the prefetch stride (Lines 11, 14), while ν specifies the prefetch depth (Lines 12, 15). The batch distances computation of neighbors follows a three-stage pipeline (Lines 13-17): 1) stride prefetch for subsequent points (Lines 14-15), 2) data access(Line 16), and 3) distance calculation with candidate set maintenance (Line 17). When $|C| \geq ef_s$, we pop farthest-point in C to keep $|C| \leq ef_s$ (Line 17). When all candidate points are processed, we apply selective re-ranking (see Section 5) to enhance the distances precision (Line 18). T is the high precision distances between points in $ANN_k(x_q)$ and x_q. Finally, the algorithm returns $ANN_k(x_q)$ with related distances T (Line 19).

B VSAG INDEX CONSTRUCTION ALGORITHM.

In *VSAG*, the insertion process for each new point x_i (Lines 3–4) can be divided into three steps. First, a search is conducted on the graph to obtain the approximate nearest neighbors $\mathrm{ANN}_k(x_i)$ and their corresponding distances T_i (Line 5). Next, pruning is performed on $\mathrm{ANN}_k(x_i)$, and the edges along with their labels are added to the graph index (Line 6). The first step, the search process, is similar to that of other graph-based index constructions. However, the second step, the pruning process, differs from other indices. Instead of performing actual pruning, *VSAG* assigns labels. Specifically, *VSAG* assigns a label $L_{i,j}$ to each edge $G_{i,j}$ in graph (see Algorithm 2). *Note that the label is used to indicate whether an edge exists under a certain construction parameter.* Given α_s, if $L_{i,j} \leq \alpha_s$, it indicates that the edge $G_{i,j}$ exists in the graph constructed using α_s or larger value as the pruning rate. During the search and pruning processes, only edges satisfying $L_{i,j} \leq \alpha_s$ need to be considered to retrieve the graph corresponding to the given construction parameters, which is proven in Theorem 4.2.

Next, for each new neighbor, reverse edges are added, and pruning is performed (Lines 7–14). If the distance $T_{i,j}$ from the current neighbor x_j to x_i is greater than the maximum distance among x_j's neighbors and x_j's neighbor list is already full (Line 8), no reverse edge is added (Line 9). Otherwise, x_i is inserted into x_j's neighbor list at an appropriate position r to maintain G_j and T_j in ascending order (Lines 10–13). Subsequently, pruning and labeling are reapplied to the neighbors of x_j located after position r. Finally, the algorithm returns the edges G and labels L of the graph index.

C PROOF OF THEOREM

C.1 Proof of Theorem 4.1

Proof. The only difference in construction under varying maximum degrees lies in the termination timing of the loop. Note that during the edge connection process, existing edges are never deleted. Specifically, when inserting x_i, the neighbors edges are continuously added to G_i until $|G_i| == a$ or $|G_i| == b$. Since $a < b$, the condition $|G_i| == a$ will be satisfied first during edge addition. In the construction process with maximum degree b, the first a edges added in the loop are identical to those in G_i^a. The subsequent $b - a$ edges inserted do not remove the first a edges. Thus, $G_i^a == G_{i,0:a}^b$. □

C.2 Proof of Theorem 4.2

Proof. We prove this by mathematical induction. Consider the insertion process of G_i^a and G_i^b, examining each point $x_j \in ANN_a(i)$ sorted by distance in ascending order.

Base Case: When inserting x_1, since both G_i^a and G_i^b are initially empty, no edges can trigger pruning, so x_1 is guaranteed to be inserted. Thus, $G_i^a = G_i^b = \{x_1\}$, and we have $G_i^a \subseteq G_i^b$.

First Iteration: Inserting x_2: We analyze the cases:

- **1: Both sets remain unchanged.** If $b \cdot dis(x_2, x_1) \leq dis(x_i, x_2)$, x_2 will not be added to G_i^b. Since $a < b$, it follows that $a \cdot$

$dis(x_2, x_1) \leq dis(x_i, x_2)$, so x_2 will also not be added to G_i^a. In this case, $G_i^a = G_i^b = \{x_1\}$.

- **2: Both sets change.** If $a \cdot dis(x_2, x_1) > dis(x_i, x_2)$, since $a < b$, it follows that $b \cdot dis(x_2, x_1) > dis(x_i, x_2)$. In this case, $G_i^a = G_i^b = \{x_1, x_2\}$.
- **3: Only G_i^b changes.** If $a \cdot dis(x_2, x_1) \leq dis(x_i, x_2)$ and $b \cdot dis(x_2, x_1) > dis(x_i, x_2)$, then $G_i^a = \{x_1\} \subset \{x_1, x_2\} = G_i^b$.
- **4: Only G_i^a changes.** If $a \cdot dis(x_2, x_1) > dis(x_i, x_2)$ and $b \cdot dis(x_2, x_1) \leq dis(x_i, x_2)$, this situation cannot occur because $a < b$. Thus, for the second insertion, we still have $G_i^a \subseteq G_i^b$.

Inductive Step: Inserting x_{n+1}: Assume that before the n-th insertion, $G_i^a \subseteq G_i^b$. We analyze the cases:

- **1: Both sets remain unchanged.** If $\exists x_j \in G_i^a$ such that $b \cdot dis(x_{n+1}, x_j) \leq dis(x_i, x_{n+1})$, then $a \cdot dis(x_{n+1}, x_j) \leq dis(x_i, x_{n+1})$ must also hold, so G_i^a remains unchanged. Since $G_i^a \subseteq G_i^b$, there must exist $x_j \in G_i^b$ such that $b \cdot dis(x_{n+1}, x_j) \leq dis(x_i, x_{n+1})$, so G_i^b also remains unchanged. Thus, $G_i^a \subseteq G_i^b$.
- **2: Both sets change.** If $\forall x_j \in G_i^b, b \cdot dis(x_{n+1}, x_j) > dis(x_i, x_{n+1})$, and $\forall x_j \in G_i^a$, $a \cdot dis(x_{n+1}, x_j) > dis(x_i, x_{n+1})$, these conditions are compatible and may occur. In this case, $G_i^a = G_i^a \cup \{x_{n+1}\} \subseteq G_i^b \cup \{x_{n+1}\} = G_i^b$.
- **3: Only G_i^b changes.** If $\exists x_j \in G_i^a \ s.t. \ a \cdot dis(x_{n+1}, x_j) \leq dis(x_i, x_{n+1})$, and $\forall x_j \in G_i^b$, $b \cdot dis(x_{n+1}, x_j) > dis(x_i, x_{n+1})$, these conditions are compatible and may occur. In this case, $G_i^a \subseteq G_i^a \cup \{x_{n+1}\} = G_i^b$.
- **4: Only G_i^a changes.** This case does not exist because Lines 5-6 of Algorithm 2 ensure that if a point is not be pruned in G_i^a, it will not be pruned again in G_i^b. Therefore, if x_{n+1} is added to G_i^a, its label is set as $L_{n+1} = a$. During subsequent iterations of α, the condition in Lines 5-6 of Algorithm 2 ensures that this point is no longer considered.

□

D SELECTING OPTIMAL ILP CONFIGURATIONS.

Under specified optimization objectives for the accuracy and efficiency of retrieval, different configurations of parameter at the index level represent candidate solutions in the design space. When environment-level and query-level parameters remain fixed, each configuration induces unique performance characteristics in terms of target metrics. The complete combinatorial space of index-level parameters constitutes the solution domain for multi-objective optimization. *VSAG*'s automated tuning mechanism aims to identify a solution subset where no configuration in the space is Pareto superior [] to any member of this subset. This set of non-dominated solution forms the Pareto Optimal Frontier [**?**], representing optimal trade-offs between competing objectives.

Through systematic evaluation of ILP configurations, *VSAG* generates performance profiles containing retrieval accuracy and latency measurements, enabling Pareto Frontier derivation. When users specify target accuracy thresholds or latency constraints, the system performs optimal configuration selection by identifying the minimal Pareto-optimal solution that satisfies the specified requirements.

Example 9. *Consider a user-defined constraint of Recall@*10 > 90% *with three Pareto Frontier candidates:* $(A, 91\%, 2000)$, $(B, 90\%, 2100)$, $(C, 89\%, 2200)$*, where tuples denote (configuration, Recall@10, queries per second). Configuration B emerges as the optimal selection, satisfying the recall threshold while maximizing query throughput through its superior QPS performance.*