CFFI Working Paper No.26-04

Constructing Parameter-Optimal Graph ANNS Index with GARENA

Puqing Wu[1], Minhui Xie[1]*, Hao Guo[2], Jie Yin[3], Sen Yang[3], Youyou Lu[2], Yunpeng Chai[1]
[1]*Renmin University of China* [2]*Tsinghua University* [3]*Tencent Inc.*

Center for Future Financial Innovation
Renmin University of China

Constructing Parameter-Optimal Graph ANNS Index with GArena

Puqing Wu[1], Minhui Xie[1]*, Hao Guo[2], Jie Yin[3], Sen Yang[3], Youyou Lu[2], Yunpeng Chai[1]
[1]*Renmin University of China* [2]*Tsinghua University* [3]*Tencent Inc.*

Abstract

Graph-based Approximate Nearest Neighbor Search (GANNS) delivers exceptional query performance compared to other algorithms, but this efficiency comes at the cost of significantly longer construction time. Because index construction parameters can alter final query throughput by up to an order of magnitude, identifying the parameter-optimal configuration is essential for real-world deployments. Yet, this process is notoriously challenging: traditional tuning methods can take days or weeks due to the need for repeated, full graph rebuilds.

This paper introduces GArena, a GANNS constructor that accelerates optimal construct parameter search. Our key observation is that while different parameter settings yield distinct graph topologies, their construction processes exhibit over 70% redundancy in distance calculations. Though a naive distance cache can eliminate this redundancy, it incurs prohibitive memory overhead (dozens of TB). Instead, GArena *redesigns the tuning paradigm by launching multiple parallel graphs with different configurations and aligning their distance computation trajectories, thereby greatly enhancing the distance cache's temporal locality.* GArena can bound the cache size to only a few MB (fully resident in CPU cache). GArena also introduces a performance-potential-guided pruner to discard unpromising configurations early on small subsets. Evaluation shows that GArena can dramatically shrink the tuning process from a day down to minutes, while consistently identifying the optimal configuration.

1 Introduction

Approximate Nearest Neighbor Search (ANNS) is the algorithmic backbone of modern AI infrastructure, powering critical systems from search engines [1] and recommender systems [2] to Retrieval-Augmented Generation (RAG) [3] for Large Language Models. Among the myriad of ANNS algorithms, graph-based methods (GANNS) [4–15] have emerged as the dominant approach due to their exceptional query efficiency and scalability. However, this query-time supremacy comes at a severe build-time cost: constructing a proximity graph requires an iterative, compute-intensive process dominated by massive high-dimensional distance calculations, often taking hours or days for large-scale datasets [16–23].

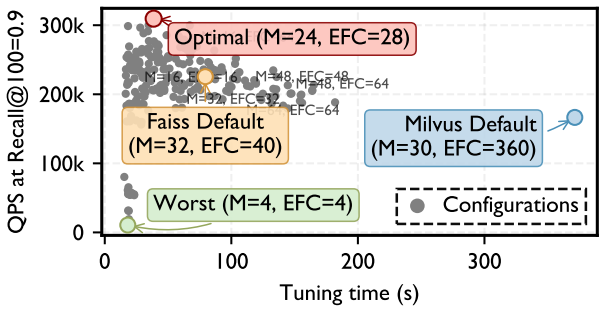

Figure 1: Performance of different construction configurations on HNSW. *Results on NSG [27] are similar.*

Unlocking GANNS's full potential requires carefully tuning construction parameters. For example, in HNSW [24], two key parameters are `M` (controlling the maximum out-degree) and `EFC` (limiting the candidate list size), which together control the quality and efficiency of the constructed index. These two parameters dictate the final query throughput by up to an order of magnitude, as shown in Figure 1. To make matters worse, the optimal parameters are highly scenario-dependent, varying drastically based on dataset characteristics (dimension/distribution/scale) and deployment requirements (top-k, recall targets, or latency-friendly vs. throughput-friendly) (see Figure 4). As a result, default parameters provided by popular libraries (e.g., FAISS [25] and Milvus [26]) are often far from optimal (1.4x–1.9x).

Achieving parameter optimality, however, is notoriously challenging. First, the relationship between construction parameters and query performance is complex and non-monotonic (Figure 3). For instance, increasing `M` typically reduces the number of hops during search, but simultane-

*Minhui Xie is the corresponding author (xieminhui@ruc.edu.cn).

ously increases the number of distance comparisons at each step. These competing effects create a rugged optimization landscape, making analytical models impossible. Second, traditional tuning methods (e.g., Bayesian Optimization like VDTuner [28]) struggle in this domain; the extreme time cost per trial makes it hard to collect enough data samples for training, which prevents learning-based models from converging effectively. Engineers thus face a painful trade-off: *either accept suboptimal empirical configurations, or pay an enormous computational bill to do an exhaustive search.* For instance, to deploy GANNS for a new dataset, engineers at Microsoft [29] need to build hundreds of distinct graph indices from scratch to identify the singular optimal configuration, consuming up to 168 hours [30] (almost a week) [28, 31].

This paper leverages two key observations to overcome these barriers. First, we identify a high percentage (over 70%) of *distance computation redundancy* during the construction process across different parameter configurations. Although the constructed graphs differ across configurations, they all digest incoming vectors in the same deterministic sequence, and inserting a vector tends to visit similar nodes in the graph. Second, we observe a *strong correlation* between a configuration's relative performance on a small data subset and its final performance on the full dataset. This implies that small subsets can serve as an effective "early-stage pruning" filter before committing to expensive full-scale construction.

Based on these insights, we present GArena, a GANNS constructor that accelerates optimal construct parameter search. GArena includes two techniques: *interval-based distance cache* and *performance-potential-guided pre-screening*.

First, GArena employs an interval-based distance cache to eliminate redundant distance computation. A naive approach is to rely on a global cache to store all computed distances and run configurations sequentially (configA → configB → configC). However, retaining all computed distances necessitates a prohibitively large cache capability (dozens of TBs). These computed distances cannot be easily evicted during construction, as they might be reused at any time whenever other configurations insert the same vector. To mitigate this, GArena redesigns the tuning paradigm by launching multiple configurations in parallel and *aligning their distance computation trajectories so that their distance computations can temporally overlap*. Specifically, GArena breaks the construction in multiple intervals and synchronizes these parallel configurations at the end of each interval, ensuring all configurations lockstep on the same set of vectors concurrently. Consequently, cached distances are consumed immediately and discarded shortly after creation, keeping the distance cache size extremely small (resident in CPU cache). We further propose a compute-ahead pipeline to achieve zero contention on this shared cache.

To further reduce total tuning time, we employ pre-screening on a small dataset subset (e.g., 5%) to filter out unpromising configurations. However, standard metrics on subsets can be misleading as some configurations degrade at scale. We introduce Performance Potential (PP), a new metric that analyzes the slope of the throughput-recall curve to predict performance potential. By combining subset performance with this predictor, GArena effectively prunes unpromising candidates before committing to expensive full constructions.

We apply GArena to two representative graph algorithms (HNSW [24] and NSG [27]). Since optimal parameters are highly scenario-dependent, we vary the deployment requirements and objective function, including different datasets, scales, top-k values, recall targets, and performance metrics, yielding a total of 28 evaluation scenarios. GArena demonstrates consistently strong performance across all these scenarios. GArena can reliably identify the optimal configuration, while cutting the tuning time by 72–95% against state-of-the-art baselines (e.g., Grid Search, VDTuner [28]). Moreover, given the same tuning time budget, GArena produces graph indices that deliver up to 1.8× higher QPS compared to VDTuner, thereby making parameter-optimal GANNS construction truly practical.

This work makes the following contributions:

- We analyze the necessity and challenges of tuning construction parameters in graph-based ANNS.
- We design GArena, an efficient parameter-optimal GANNS index constructor. GArena introduces two techniques: an interval-based distance cache that exploits distance computation redundancy to maximize reuse, and a performance-potential-guided pruner to discard unpromising configurations on small subsets.
- We conduct extensive experiments on representative datasets, demonstrating that GArena significantly outperforms state-of-the-art tuning methods.

2 Background and Motivation

2.1 Graph-based ANNS Construction Process

Among various ANNS indexes, graph-based methods [4–9] are leading and widely-used approaches due to their superior search efficiency. These methods construct a proximity graph where each node represents a vector, and edges connect nodes that are close to each other in the vector space. During querying, the search process traverses the graph by iteratively moving to the neighbor closest to the query vector. Owing to the small-world property [24] of the graph, this structure enables rapid navigation and allows nearest neighbors to be located with logarithmic complexity.

While graph-based indices deliver great query performance, they suffer from prohibitively long construction time compared to other types of indexes [17, 32]. Even for medium-scale datasets (e.g., 100 million vectors), constructing a graph index can take several hours. The construction of graph-based ANNS indexes is an iterative and incremental process; see Figure 2. Vectors are inserted one by one into the graph, and

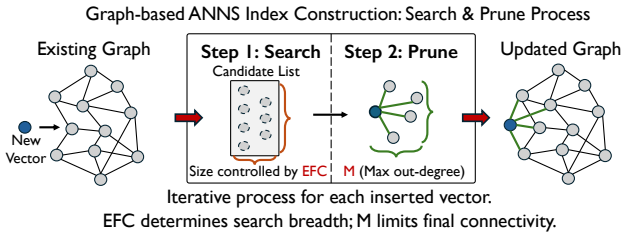

Figure 2: Graph-based ANNS index construction process.

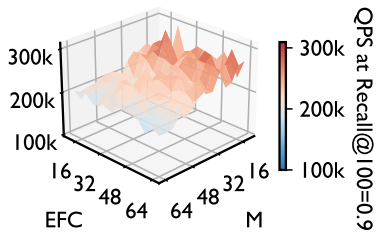

Figure 3: Non-monotonic relationship and interaction between `M` and `EFC` parameters.

for each incoming vector, the system needs to determine its optimal position within the existing graph, which involves two steps: 1) *Search*: traversing the currently built graph by treating the new vector as a query to identify a pool of candidate neighbors for it, and 2) *Prune*: selecting the final set of neighbors from the candidate list. The quality of the index is determined by construction parameters that control the graph's connectivity. Although different graph algorithms define their parameter sets, these parameters play analogous roles. For example, HNSW [24] uses `M` (maximum out-degree) and `EFC` (candidate list size), while NSG [27] uses `R` (maximum out-degree) and `L` (candidate list size). To simplify presentation, we use `M` and `EFC` as the default annotations.

Distance calculation as the primary bottleneck. In the whole construction process, our performance profiling reveals that distance calculation is the unequivocal primary performance bottleneck, typically accounting for over 90% of the total construction time [32]. This bottleneck arises because: 1) each individual distance calculation has a time complexity of $O(d)$, where d is the vector dimensionality (typically hundreds to thousands), and 2) the construction process requires executing the search step for every vector insertion, involving numerous distance calculations as the algorithm traverses the graph. For large-scale datasets, the total number of distance calculations becomes enormous, making distance computation the primary source of construction time.

2.2 Parameter Tuning for Graph-based ANNS

Importance of construction parameter selection. As introduced in §2.1, construction parameters (e.g., `M`/`EFC` for HNSW, `R`/`L` for NSG) play a crucial role in determining index quality and query performance. To empirically demonstrate this impact, we conduct a comprehensive Grid Search on the SIFT-10M dataset, sweeping `M` from 4 to 64 and `EFC` from 4 to 360. For each configuration, we build a corresponding HNSW index and evaluate its *QPS rate at recall@100=0.9*. Unless otherwise specified, all motivation and insight figures in this section and §2.3 use this default setting; evaluation figures use the same default (see §4).

Figure 1 shows the results. Our results reveal a massive performance variance: the optimal configuration (marked with a red circle) achieves a QPS rate that is significantly higher than that of suboptimal configurations. Moreover, default parameters from popular libraries such as FAISS [25] (marked with a yellow circle) and Milvus [26] (marked with a blue circle) trailed the optimal QPS by over 46%, demonstrating that careful parameter selection is essential and default settings recommended by libraries fail to unlock the full potential of the graph index.

No one construction configuration can fit all scenarios. Our analysis demonstrates that optimal parameters are highly scenario-dependent and sensitive to four deployment dimensions: datasets, top-k, recalls, and performance metrics. A static parameter configuration cannot adapt to the diverse requirements of real-world applications. Figure 4 shows how optimal construction parameters vary across these dimensions; each bar represents a distinct configuration, with the optimal highlighted in red. We observe that: 1) *Sensitivity to dataset*: different datasets (SIFT-10M vs. DEEP-10M) favor different parameter configurations due to distinct dimensionality and data distribution characteristics. 2) *Sensitivity to top-k/recall*: workloads requiring high precision (e.g., 0.99) or large top-k (e.g., top-k =1000) demand higher-quality graph structures, directly influencing optimal parameter selection. Parameters efficient for easy targets often fail strict requirements, while parameters for strict targets are wasteful for easier tasks. 3) *Sensitivity to performance metrics*: different system objectives (e.g., maximizing throughput vs. minimizing latency) lead to divergent optimal parameters. This multi-dimensional sensitivity makes parameter selection a moving target, rendering static or default configurations ineffective.

Complex parameter interdependencies. Identifying these shifting optima is further complicated by the non-monotonic and interdependent relationship between `M` and `EFC`. As shown in Figure 3, construction parameters exhibit non-monotonic relationships with the final query performance. This arises because both `M` and `EFC` exhibit complex trade-offs: `M` reduces search hops through denser connections but increases per-hop cost beyond a certain threshold, while `EFC` enables more accurate neighbor selection but exhibits diminishing returns.

Moreover, the parameters are coupled: the optimal value of `M` depends on the `EFC` setting, and vice versa, creating a three-dimensional optimization landscape. This interdependency means that tuning one parameter while keeping the other fixed may not yield globally optimal results.

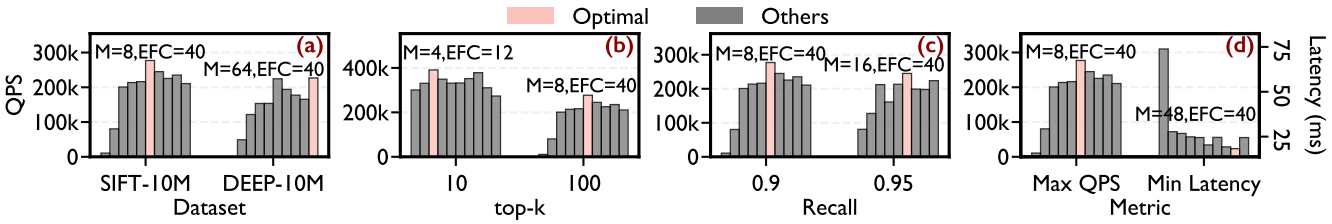

Figure 4: Scenario-dependent optimal construction parameters. *Each bar:* $M \in \{8, 16, 32, 48, 64\}$, $EFC \in \{40, 80\}$. *NSG follows a similar trend, and its results are omitted for space.*

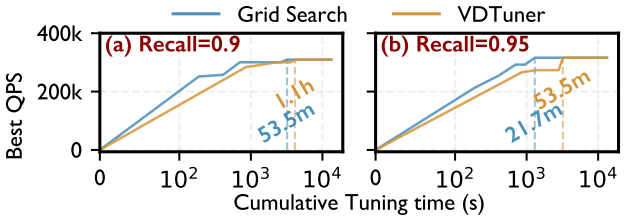

Figure 5: Tuning time comparison of Grid Search and VDTuner. *Vertical dashed lines mark the turning point (the time when each strategy first reaches its final best QPS).*

Prohibitive cost of conventional tuning methods. Conventional parameter tuning methods struggle in this setting because evaluating a single configuration (i.e., a trial) entails fully constructing the index from scratch (§2.1) and then issuing a substantial number of queries to estimate its QPS. Here, we examine two representative methods: Grid Search and VDTuner [28] (a representative Bayesian Optimization-based tuner). Grid Search guarantees finding the global optimum within a discretized space, but it requires evaluating all combinations of `M` and `EFC` candidates. VDTuner reduces trials via learning-model-guided sampling. Figure 5 plots the best throughput found versus cumulative tuning time. Experiments are conducted under the default setting, sweeping `M` and `EFC` from 4 to 64.

Although Grid Search theoretically guarantees finding the global optimum configuration, it suffers from a combinatorial explosion. Given that a trial can take several minutes even with 384 threads in this case, exhaustively evaluating the full Cartesian product is computationally costly.

VDTuner is often preferred for expensive tuning tasks due to its high sampling efficiency. However, for ANN index construction tuning, VDTuner yields limited benefits compared to Grid Search. Figure 5 marks the *turning point* with the vertical dashed line, which is the earliest time when a method first reaches the best performance it will ever achieve during the whole tuning run. VDTuner reaches this point later in our results (e.g., at recall=0.9, it requires 53.5 min for Grid Search and 68.9 min for VDTuner), suggesting that learning-based methods may be more fragile under limited budgets in this setting. This limited effectiveness stems from the scarcity of observation data: unlike query-parameter tuning where thousands of configurations can be tested in seconds, index construction takes several minutes per trial. With only a handful of feasible observations (e.g., 10 trials in an hour), the Bayesian surrogate model remains highly uncertain, preventing convergence and exhibiting limited guidance capability.

In a short conclusion, existing parameter tuning methods are ill-suited for ANNS construction due to the unique challenges of this domain, which features a relatively small search space but an extremely high computational cost per trial. These approaches are fundamentally constrained by two factors: 1) Black-box perspective: They treat the index builder as an opaque function, necessitating a complete, time-consuming index rebuild for every configuration tested. 2) Data scarcity for learning: The sample scarcity makes learning-based approaches lack sufficient training samples to fit the objective function or effectively guide the search accurately.

2.3 Opportunities and Challenges

To overcome the prohibitive cost of parameter tuning, we conduct an in-depth analysis of the index construction process. Our investigation reveals two key observations (computational redundancy and cross-scale predictability) that motivate the design of GArena.

Opportunity 1: High computation redundancy across different configurations. As introduced in §2, distance calculation accounts for over 90% of the total construction time. We observe that when constructing indices under different parameter configurations, a significant portion of these distance calculations are redundant. As shown in Figure 6, the heatmaps visualize the overlap ratio of distance computations across different configuration pairs on SIFT-10M for two representative graph algorithms: HNSW (Figure 6a) and NSG (Figure 6b). Both algorithms exhibit over 70% overlap, with NSG reaching up to 92%. We find that the reason is that, regardless of the specific parameter settings, the graph construction algorithm inserts vectors in the same sequential order (based on vector ID). Consequently, the search path for a newly inserted vector v_i often traverses similar regions of the graph, necessitating distance comparisons against the similar set of neighbors v_j. This high redundancy indicates

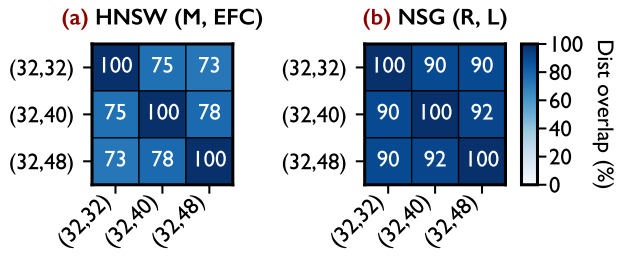

Figure 6: Distance computation overlap across configurations on two GANNS indexes.

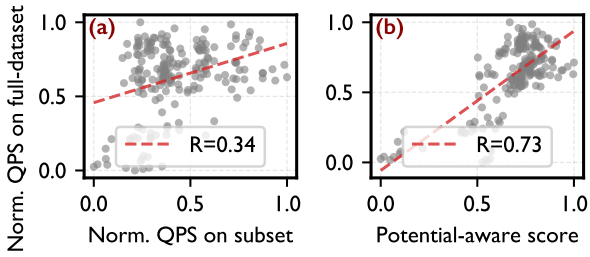

Figure 7: Predictability via small-scale subsets. *(a) Norm. QPS on subset vs. norm. QPS on full-dataset (Pearson's r=0.34). (b)* GARENA*'s potential-aware score vs. norm. QPS on full-dataset (Pearson's r=0.73).*

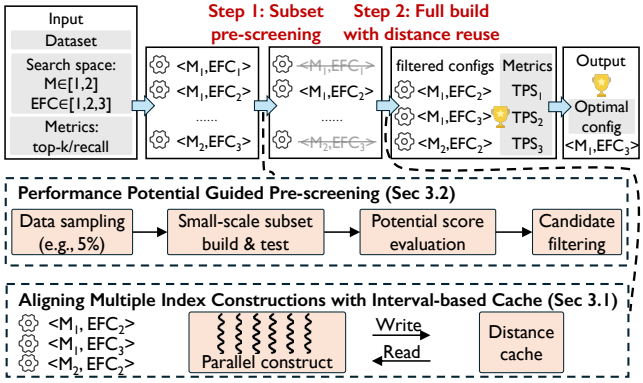

Figure 8: GARENA overview.

that distance computations should be cached and reused. By treating distance computations as a reusable asset, we can theoretically amortize the dominant cost of index construction across all trials.

Challenge: Implementing this reuse in practice presents a severe memory challenge. A naive implementation of distance cache that stores all computed distances would require $O(n \log n)$ memory space, where n denotes the dataset capacity and the $\log n$ term reflects that each vector insertion triggers $O(\log n)$ distance computations in typical graph-based indexes. For large-scale datasets, the cache size would rapidly escalate to dozens of TB, far exceeding the memory capacity of standard servers.

Opportunity 2: Predictability via small-scale subsets. Our second insight challenges the assumption that parameter configurations must always be evaluated on the full dataset to determine their quality. To quantify this, we evaluated configurations on a small subset (e.g., 5% of the dataset) and compared their relative (normalized) QPS against that on the full dataset. Figure 7a visualizes the correlation: each point (x, y) represents a configuration, where x denotes its normalized QPS on the subset and y denotes that on the full dataset. This correlation enables inexpensive pre-screening on the subset to prune candidates early. However, subset QPS alone is an imperfect proxy. We quantified this naive correlation by computing the Pearson correlation coefficient [33], yielding a correlation but a weak correlation of $r = 0.34$.

Challenge: While a naive correlation exists, it is not perfect; some configurations that perform well on a small-scale subset degrade rapidly when the data scale increases (the top-left corner of Figure 7a). However, this degradation is not random. We identify that by analyzing the sensitivity of a configuration on the subset (specifically, how its throughput changes in response to stricter recall targets), we can predict its robustness to data scaling. This allows us to define a Performance Potential (PP) metric to greatly enhance this correlation, yielding a strong Pearson correlation coefficient ($r = 0.73$); see Figure 7b.

Together, these two opportunities define the design philosophy of our proposed system, GARENA. Instead of blindly iterating through configurations, GARENA exploits computation reuse to lower the cost per trial and employs subset-based prediction to minimize the number of full trials required.

3 GARENA Design

Figure 8 shows the process overview of GARENA. GARENA takes the target dataset, a predefined parameter search space, and the user's target metrics (e.g., top-k, recall, and TPS).

- **Step 1: Subset pre-screening.** Instead of evaluating all configurations on the full dataset, GARENA first samples a tiny subset (e.g., 5%). It builds and tests all candidate configurations on this small scale. Using a *performance potential predictor* (§3.2), GARENA aggressively filters out unpromising and "brittle" candidates. Only the robust configurations proceed to the next stage.
- **Step 2: Full build with distance reuse.** For the surviving configurations, GARENA performs a full-scale index construction. GARENA constructs them in parallel and lets them share an *interval-based distance cache* (§3.1). By aligning the insertion trajectories of different configurations, the cache temporal locality is greatly enhanced, and its memory footprint is bounded to only MBs.

After the accelerated full builds, GARENA evaluates the final metrics for the remaining configurations and outputs the optimal configuration (and the graph index) that perfectly meets the user's requirements.

3.1 Interval-based Distance Cache

In this section, we first analyze the limitations of a naive distance cache, then introduce an interval-based mechanism that aligns multiple constructions at the interval granularity to achieve memory efficiency, and finally present a compute-ahead optimization to remove all contention in the cache.

3.1.1 Naive Distance Cache

Observing that more than 70% of the distance calculations are redundant, a straightforward solution is to implement a global distance cache. Conceptually, this cache materializes and stores *all* computed distance results $dist(i, j)$ for vector pairs (ID i and j) during construction, and reuses them for all subsequent configurations.

While highly effective for small-scale data, a naive global cache would hit the "memory capacity wall" for large-scale datasets. For instance, on a dataset with 100M vectors, a naive global cache that stores all distance results generated during index construction would require tens of TBs prohibitively.

Root cause: long lifespan for cached entries. We analyze that the demand for large-capacity caches fundamentally arises from the extremely long lifespan of cached entries inherent in the traditional sequential parameter tuning paradigm (i.e., configA → configB → configC). Here, the lifespan refers to the time duration between when a distance value is computed and when it can be reused across different configurations. In this sequential mode, distance information generated for a specific vector v during the construction of config A can only be reused when config B/C processes the same vector v afterwards. Since config B only starts after config A is fully completed, the stored distance must be preserved in cache throughout the entire duration of the intermediate construction process. The graph index construction process consists of n vector insertions, and each vector requires $O(\log n)$ distance computations during insertion [24]. Therefore, the cache memory consumption grows at $O(n \log n)$ with dataset size n.

3.1.2 Interval-based Cache: Aligning Multiple Constructions with Interval-based Synchronization

As analyzed in §2.3, making in-memory caching feasible requires drastically shortening the lifespan of cached entries. Our key idea is to ***align*** **the construction trajectories of multiple configurations so that their distance computations for the same vectors can temporally overlap**, allowing cached distances to be reused and discarded soon after they are produced, rather than being kept alive for the entire build.

We realize this idea through an interval-based distance cache design that fundamentally reduces memory consumption by limiting cache scope to an *interval*. We define an *interval* as a contiguous sequence of vectors. Specifically, instead of processing the vector dataset as a monolithic block, GArena divides the dataset into contiguous intervals of size W, where the i^{th} interval contains vectors with IDs from $(i-1)W+1$ to iW. The cache is ephemeral: it only stores computed distances relevant to the current active interval. Once the system advances to the next interval, cached data from the previous interval is discarded. All configurations are constructed in a **lockstep manner**, progressing at the same pace across intervals and reusing distance computations generated within the same interval. This design ensures that cached distances are reused shortly after they are produced, thereby bounding the lifespan and significantly reducing cache capacity requirements.

Feasibility of interval-based cache. One might think that an interval-based cache may not yield enough cache hits within a short time window (interval) before being discarded. Here, we demonstrate why this design is not only feasible but highly efficient. We observe that existing graph-based index construction algorithms commonly follow a sequential and iterative construction paradigm, in which vectors are inserted into the graph one by one in a fixed order. For any distance computation $dist(v_i, v_j)$, although differences in construction parameters affect the resulting graph topology (such as the number of neighbors explored or the final number of edges connected, i.e., variations along the j dimension), they do not alter the order in which vectors are processed (i.e., the i dimension order). This property guarantees that if we align multiple constructions, they will require similar distance computations for the same target vectors v_i at approximately the same time, enabling efficient reuse within each interval.

Parallel construction and interval-level aligning. To exploit interval-based caching across configurations, GArena constructs multiple parameter configurations concurrently and enforces that they advance through the dataset in lockstep at the interval granularity. As illustrated in Figure 9, GArena launches the construction of T configurations concurrently and keeps them aligned at the granularity of intervals. Within each interval, all configurations process the same set of vectors, so distance computations for a given vector are issued by different configurations within a short time window. Once a distance for that vector has been computed by one configuration and inserted into the cache, the remaining configurations that reach the same query within the interval can simply read it from the cache instead of recomputing it. Overall, a larger T increases reuse within each interval but also raises memory consumption, because each additional configuration requires maintaining another partially built graph in memory. In our implementation, we choose the largest T before out-of-memory.

To enforce this interval-level alignment, GArena places a synchronization barrier at the end of each interval. No configuration is allowed to proceed to $(k+1)^{th}$ interval until every configuration has finished inserting all vectors in the k^{th} interval. This guarantees that all cached distances produced in the k^{th} interval remain available until the slowest configuration

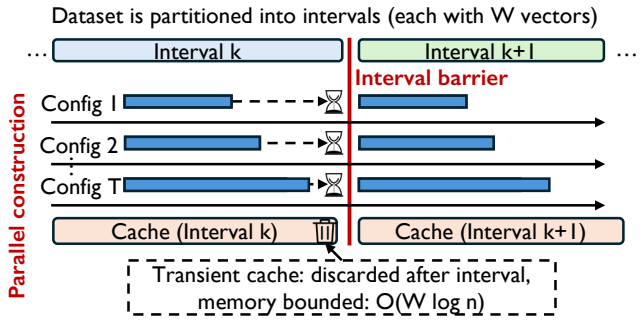

Figure 9: Align multiple index constructions with interval barriers.

has finished that interval.

Advantages. Our interval-based design brings two advantages. First, the lifespan of cache entry is significantly shortened, and the entire distance cache can reside in main memory, avoiding the prohibitive latency of out-of-core (e.g., SSD) accesses. Since each interval contains W vectors and each vector insertion triggers $O(\log n)$ distance computations, the size of distance cache is reduced from $O(n \log n)$ to $O(W \log n)$, where $W \ll n$, enabling the entire cache to fit in main memory even for billion-scale datasets.

Second, with a small W, the distance cache can be kept CPU cache resident, further improving efficiency. We choose W such that the distance cache resides in CPU cache (we set $W = 100$ by default, where only dozens of MB suffice for billion-scale datasets). On a typical server CPU, recomputing a distance with AVX-512 costs ~150 ns, while a CPU cache hit for a cached distance is on the order of 10 ns. By keeping the working set resident in CPU caches, interval-based caching makes reuse ultra-fast compared to recomputing.

3.1.3 Contention-Free Compute-Ahead Pipeline

While interval-based alignment improves cache locality, it also introduces new concurrency challenges on the distance cache. We first quantify the overhead caused by cache contention and reveal a key trade-off between contention cost and recomputation cost that drives our designs. We then introduce two targeted optimizations that eliminate both write–write and read–write conflicts.

High cost of contention. In a naive implementation, all construction tasks operate in a mixed read/write mode: for each distance calculation, a thread first probes the shared distance cache, and upon a miss, computes the distance and writes it back. In a high-concurrency setting, multiple configurations will concurrently access shared cache entries, causing contention. Contrary to the intuition that cached reads are always cheaper than recomputation, we find that in high-concurrency scenarios, synchronizing read/write operations on a shared distance cache under contention can easily cost hundreds of ns, even exceeding the cost of recomputing a distance with AVX-512 (~150 ns). Contended writes are even worse: when multiple configurations write to the same cache line, they may trigger severe coherence traffic, further inflating latency.

Based on these observations, we establish a core design principle: *It is better to sacrifice a marginal amount of reuse (by recomputing) than to incur the high latency of synchronization conflicts.* Consequently, our architecture prioritizes zero-contention over maximum hit rate. GArena introduces a single-writer architecture and dual cache with compute-ahead optimizations to eliminate write–write and read–write conflicts, respectively.

Single-writer/multi-reader architecture. Following this design philosophy, we first simplify the shared distance cache read/write pattern with a single-writer/multi-reader model. We designate the configuration with the longest build time as the *writer* (in practice, configurations with larger `M` and `EFC` values tend to take longer to build and perform more distance computations), while all other configurations act as *readers*. The writer alone is responsible for populating the distance cache. All readers read the shared distance cache for distance results, and if a cache miss occurs, the reader computes the distance locally but does not write it back to the shared distance cache.

Fundamentally, this design restricts reuse such that all configurations consume distance results only from this single superset configuration (the writer). We acknowledge that permitting multiple configurations (e.g., the top-3 most complex ones) to write to the cache would theoretically increase the cache hit ratio, as it would capture a wider union of calculated distances. However, our analysis shows that this approach is counter-productive: the performance penalty introduced by multiple threads competing for write access to the shared cache negates any gains achieved from the incremental reduction in distance computations.

It is also important to clarify that the writer task is naturally often executed by multiple parallel threads, but this thread-level parallelism does not reintroduce contention. We exploit the inherent data parallelism of graph construction: each thread is responsible for inserting a distinct vector at any time. During the insertion of a vector v_i, the algorithm computes distances $dist(i, j)$ to many candidate neighbors v_j. Our cache is organized as a two-level hash keyed by (i, j): first by the source ID i and then by the target ID j. Threads that handle different i write to distinct top-level buckets, so their write paths are disjoint and never collide. Thus, writes to the cache remain contention-free even without locks.

Dual cache with compute-ahead optimization. The formermentioned single-writer architecture eliminates write-write conflicts. We further propose a compute-ahead pipeline that eliminates read-write conflicts. Specifically, GArena simultaneously maintains two distinct distance caches:

- Current distance cache (C_{curr}, read-only): The populated distance for the active interval k. All readers read this cache

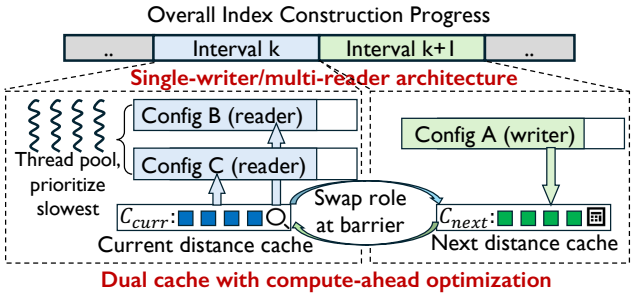

Figure 10: Contention-free compute-ahead pipeline.

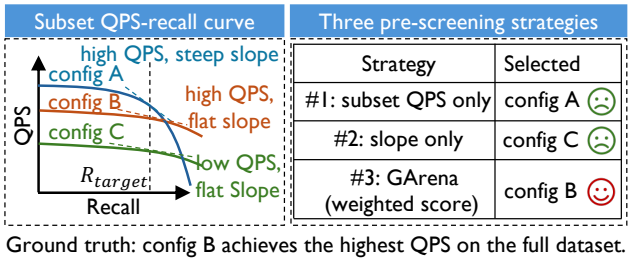

Figure 11: Performance potential guided pre-screening.

to perform graph construction for interval k.

- Next distance cache (C_{next}, write-only): A write-only cache for the upcoming interval $k+1$. This cache is populated ahead of time by the writer threads.

The pipelined workflow is as follows. While reader tasks read distance entries from C_{curr} to perform graph construction for interval k, the writer task engages in a *compute-ahead* process: it inserts vectors from interval $k+1$, and records all computed distances into C_{next}. Since readers only read C_{curr} and the writer only writes to C_{next}, their read and write operations never target the same cache entries simultaneously. At the end of each interval, the roles of the two caches are swapped atomically.

To minimize idle time at the barrier, GARENA balances the progress of readers and writer. Specifically, GARENA maintains two task queues storing vector IDs to be inserted into the graphs in the active interval, a reader queue for the current interval k, and a writer queue for the next interval $k+1$. The scheduler always services the lagging queue:

- When processing a vector v_i from the reader queue, a thread performs a batched insertion by inserting v_i into the graphs of all reader configurations sequentially. This batching amortizes the cost of fetching distance entries from C_{curr}: the thread loads the corresponding $(i,*)$ bucket into its core-private L1/L2 cache once and reuses it for every reader configuration, improving cache locality and reducing memory stalls.
- When processing a vector v_i from the writer queue, it executes the compute-ahead logic. It inserts v_i into the graph of the writer configuration and caches all computed distances in C_{next}.

Overall, the single-writer/multi-reader model and the dual-cache compute-ahead pipeline together yield a contention-free distance cache. They deliberately trade a small amount of potential reuse for drastically lower synchronization overhead, which empirically results in faster end-to-end construction.

3.2 Performance Potential Guided Pre-screening

While the interval-based cache (§3.1) reduces the cost of each full-dataset build per configuration, the search space for optimal configurations remains vast. To further accelerate tuning, we introduce a pre-screening step. The core idea is to evaluate candidate configurations on a small fraction of the dataset (e.g., 5%) to prune candidates early, thereby reserving expensive full-dataset builds for only promising ones.

A counterintuitive phenomenon emerges when tuning parameters on small subsets: configurations that achieve the highest throughput on the subset (we call it *subset performance, SP*) often fail to maintain their performance when scaled to the full dataset. This observation challenges the straightforward approach of selecting parameters solely based on their performance on a small subset in isolation.

To overcome the inadequacy of subset performance in characterizing performance potential, we propose a new metric named *Performance Potential (PP)*. Our key insight is that the *slope* of the throughput–recall curve around the target recall on the subset is highly indicative of how well a configuration will sustain its performance when scaled to the full dataset. Figure 11 illustrates an example. If a configuration experiences a sharp drop in throughput to achieve a marginal gain in recall (e.g., config A), it indicates the graph structure is brittle. It is likely over-optimized only for the current scale and lacks the search capability required to handle the increased complexity of a much larger dataset. However, configurations (e.g., config B) that maintain stable throughput across a range of recall targets imply a robust graph structure. This elasticity suggests the configuration is more likely to sustain its performance when the dataset size expands by orders of magnitude.

Formally, we define the Performance Potential metric as the magnitude of the gradient of the throughput-recall curve at the user's target recall (R_{target}). Mathematically:

$$\text{Performance Potential} = \left| \frac{\partial Throughput}{\partial Recall} \right|_{Recall=R_{\text{target}}}$$

Since throughput decreases as recall increases, the slope is negative, and we take its absolute value to obtain a non-negative metric. In practice, we estimate this local gradient using weighted finite difference methods [34], by evaluating multiple recall-throughput points surrounding R_{target}.

To validate the predictive power of our PP metric, we conduct a correlation analysis comparing how well subset metrics predict full-dataset performance. We evaluate parameter configurations ($\texttt{M} \in [4,64]$, $\texttt{EFC} \in [4,64]$ with an interval of 4)

on both the 5% subset and the full SIFT-10M dataset. Figure 7 shows the correlation between subset metrics and the ground-truth QPS on the full dataset. The naive approach, which relies solely on subset QPS, exhibits a weak correlation with full-dataset performance (Pearson's $r = 0.34$, weak). In contrast, our PP metric demonstrates a higher correlation with the actual full-dataset QPS (Pearson's $r = 0.73$, strong).

Combining subset performance with performance potential. Relying exclusively on PP, however, might inadvertently favor highly stable but inherently sluggish configurations. Therefore, to comprehensively evaluate and prune candidates, GArena employs a dual-dimensional pruning method that balances both immediate subset effectiveness (SP) and scalability prediction (PP). SP measures the parameter's throughput on the subset, directly reflecting immediate effectiveness. PP, as defined above, predicts the parameter's ability to maintain performance when scaled to the full dataset. Filtering based solely on SP fails to eliminate fast but brittle configurations, whereas relying only on slope stability fails to exclude stable yet sluggish ones.

To combine these two dimensions for comprehensive parameter evaluation, we use a weighted scoring function:

$$\text{Score} = \alpha \cdot \mathcal{N}(\text{SP}) + (1-\alpha) \cdot (1 - \mathcal{N}(\text{PP}))$$

where $\mathcal{N}(\cdot)$ denotes the min-max normalization function, and α is a tunable trade-off parameter ($0 \le \alpha \le 1$) used to balance SP and PP. Since a smaller PP value indicates better Performance Potential, we use $(1 - \mathcal{N}(\text{PP}))$ in the scoring function. The choice of α depends on the specific requirements of the application. Through empirical analysis, we recommend $\alpha = 0.5$ as a balanced default.

This scoring mechanism serves as a pre-screening filter: only the top-ranked candidates (top 50% in our implementation), which demonstrate both a strong baseline and high performance potential, are advanced to the full-scale index construction stage, while the rest are pruned.

It is also important to note that the computational overhead of scores is negligible compared to full-dataset index construction. For calculating both SP and PP, it requires only a single index construction per configuration on the small subset and multiple lightweight searches to obtain the necessary throughput-recall data points to calculate the slope. Since common graph-based index construction has a time complexity of $O(n \log n)$, where each vector insertion triggers $O(\log n)$ distance computations, for a subset containing 5% of the data, the construction time roughly scales with the subset size (up to a logarithmic factor), and is therefore substantially smaller than the full-dataset construction time.

Limitation. Our pre-screening is inherently heuristic. In certain edge cases, the true global optimal configuration might temporarily underperform on the subset and be mistakenly pruned before the full-scale build. In practice, users can tune the pre-screening rate to flexibly trade off search time against optimality. If an application demands absolute peak performance at all costs, users can simply disable the pre-screening (i.e., GArena w/o Pruning in §4), guaranteeing that GArena always discovers the optimal parameter configuration.

4 Evaluation

4.1 Experimental Setup

The experiments are conducted on a server equipped with two AMD EPYC 9654 96-Core processors (192 physical cores, 3.7 GHz) and 1TB memory. The server runs Ubuntu 24 with kernel version 6.14.0-37-generic.

Implementation. We implement GArena in C++ (~1,500 lines of code) and seamlessly integrated it into two representative and widely deployed GANNS indices, HNSW [24] and NSG [27]. For both indices, the integration involves three core architectural modifications: ①*Distance cache hook*: We wrap the original distance computation function with a caching layer that intercepts every distance call, replacing redundant recomputations with hash-table lookups. ②*Parallel construction*: We replace the sequential index-building loop with a single-writer/multi-reader thread pool that processes vectors at interval granularity, where the writer populates the shared distance cache and readers consume it to build indexes with different configurations concurrently. ③*Pre-screening on subsets*: We add a lightweight pre-build stage that constructs indexes on a small subset and scores each configuration; low-scoring configurations are pruned early without full-dataset evaluation. With a clean and non-intrusive API, GArena is very easy to integrate to HNSW and NSG. We will open-source our codebase.

Competitors. We compare GArena with the following parameter tuning methods:

- Grid Search: Exhaustively evaluates all parameter configurations to find the optimal configuration.
- VDTuner [28]: A state-of-the-art black-box Bayesian optimization method dedicated to parameter tuning for ANNS. Note that Milvus [35] paper also states they adopt Bayesian optimization to tune construction parameters, but this module is not open-sourced.

Variants of GArena. We evaluate GArena in two variants to isolate the contribution of each component. GArena w/o Pruning uses only the interval-based distance cache, while GArena w/ Pruning enables both the cache and performance-potential-guided pre-screening.

Datasets. We use three representative datasets:

- SIFT-10M: 10 million 128-dimensional vectors [36].
- DEEP-10M: 10 million 96-dimensional vectors [37].
- SIFT-1B: 1 billion 128-dimensional vectors [36].

Metrics. To evaluate the robustness of GArena, recognizing that the definition of optimal shifts dramatically based on user requirements, we vary the deployment requirements and objective function, including different in-

Dataset	Metric	top-k	Recall	Grid Search		GArena w/o Pruning		GArena w/ Pruning		VDTuner (Budget 1)		VDTuner (Budget 2)	
				Perf	Time	Perf	Time	Perf	Time	Perf	Time	Perf	Time
SIFT-10M	QPS	100	0.9	309.5k	3.7h	309.5k	25.7m	309.5k	13.7m	291.2k ↓	25.7m	284.2k ↓	13.7m
	QPS	10	0.9	413.5k	3.7h	413.5k	29.3m	413.5k	15.1m	361.5k ↓	29.3m	341.1k ↓	15.1m
	QPS	1000	0.9	25.7k	3.7h	25.7k	29.3m	25.7k	19.7m	21.2k ↓	29.3m	21.2k ↓	19.7m
	QPS	100	0.85	355.7k	3.7h	355.7k	19.8m	355.7k	10.5m	315.3k ↓	19.8m	315.3k ↓	10.5m
	QPS	100	0.95	316.1k	3.7h	316.1k	19.8m	316.1k	12.2m	266.2k ↓	19.8m	266.2k ↓	12.2m
	QPS	100	0.99	266.4k	3.7h	266.4k	22.7m	248.4k ↓	16.1m	214.1k ↓	22.7m	214.1k ↓	16.1m
	Latency (ms)	100	0.9	16.6	3.7h	16.6	45.2m	17.3 ↓	5.7m	17.3 ↓	45.2m	21.2 ↓	5.7m
DEEP-10M	QPS	100	0.9	229.0k	3.2h	229.0k	14.0m	228.7k ↓	5.0m	218.1k ↓	14.0m	218.1k ↓	5.0m
	QPS	10	0.9	286.9k	3.2h	286.9k	10.4m	286.9k	7.6m	269.4k ↓	10.4m	269.4k ↓	7.6m
	QPS	1000	0.9	32.5k	3.2h	32.5k	30.2m	32.5k	22.8m	27.0k ↓	30.2m	27.0k ↓	22.8m
	QPS	100	0.85	255.9k	3.2h	255.9k	30.2m	255.9k	19.7m	233.7k ↓	30.2m	233.5k ↓	19.7m
	QPS	100	0.95	223.4k	3.2h	223.4k	18.0m	209.2k ↓	12.7m	222.1k ↓	18.0m	222.1k ↓	12.7m
	QPS	100	0.99	191.5k	3.2h	191.5k	14.0m	191.5k	14.0m	188.5k ↓	14.0m	188.5k ↓	14.0m
	Latency (ms)	100	0.9	33.1	3.2h	33.1	53.2m	33.1	40.2m	33.8 ↓	53.2m	35.7 ↓	40.2m

Table 1: Tuning time and performance comparison on HNSW. *Grid Search serves as the baseline.* ***Perf Color:*** *black: performance equals baseline, red ↓: performance degradation compared to baseline.* ***Budgets:*** *For each setting, Budget 1 (resp. Budget 2) gives VDTuner the same wall-clock tuning time as* GArena *w/o Pruning (resp.* GArena *w/ Pruning).*

Dataset	Metric	top-k	Recall	Grid Search		GArena w/o Pruning		GArena w/ Pruning		VDTuner (Budget 1)		VDTuner (Budget 2)	
				Perf	Time	Perf	Time	Perf	Time	Perf	Time	Perf	Time
SIFT-10M	QPS	100	0.9	124.4k	7.8h	124.4k	45.6m	113.6k ↓	30.2m	78.9k ↓	45.6m	78.9k ↓	30.2m
	QPS	10	0.9	272.2k	7.8h	272.2k	45.7m	272.2k	26.7m	215.1k ↓	45.7m	215.1k ↓	26.7m
	QPS	1000	0.9	20.6k	7.8h	20.6k	45.2m	20.6k	25.7m	12.3k ↓	45.2m	12.3k ↓	25.7m
	QPS	100	0.85	129.3k	7.8h	129.3k	45.4m	129.3k	26.0m	78.8k ↓	45.4m	78.8k ↓	26.0m
	QPS	100	0.95	126.0k	7.8h	126.0k	45.5m	111.7k ↓	26.5m	80.9k ↓	45.5m	80.9k ↓	26.5m
	QPS	100	0.99	127.8k	7.8h	127.8k	45.8m	94.9k ↓	29.6m	79.6k ↓	45.8m	79.6k ↓	29.6m
	Latency (ms)	100	0.9	29.7	7.8h	29.7	45.2m	29.7	31.7m	31.5 ↓	45.2m	31.5 ↓	31.7m
DEEP-10M	QPS	100	0.9	149.0k	8.2h	149.0k	50.7m	149.0k	30.4m	83.5k ↓	50.7m	83.5k ↓	30.4m
	QPS	10	0.9	303.4k	8.2h	303.4k	51.4m	303.4k	30.2m	236.0k ↓	51.4m	236.0k ↓	30.2m
	QPS	1000	0.9	22.1k	8.2h	22.1k	34.0m	22.1k	14.1m	14.1k ↓	34.0m	14.1k ↓	14.1m
	QPS	100	0.85	155.7k	8.2h	155.7k	43.6m	155.7k	21.9m	90.0k ↓	43.6m	90.0k ↓	21.9m
	QPS	100	0.95	152.4k	8.2h	152.4k	51.8m	152.4k	31.3m	88.1k ↓	51.8m	88.1k ↓	31.3m
	QPS	100	0.99	153.6k	8.2h	153.6k	51.4m	128.7k ↓	13.3m	86.3k ↓	51.4m	86.3k ↓	13.3m
	Latency (ms)	100	0.9	28.6	8.2h	28.6	44.3m	28.6	21.3m	30.9 ↓	44.3m	30.9 ↓	21.3m

Table 2: Tuning time and performance comparison on NSG. *Same format as Table 1.*

dexes (HNSW/NSG), datasets (SIFT/DEEP), top-k values (10/100/1000), recall targets (0.85/0.9/0.95/0.99), and performance metrics (QPS/latency-oriented), yielding a total of 28 deployment scenarios. For each scenario, we report the overall tuning time (i.e., constructing the parameter-optimal index) and the quality of the best constructed index measured by throughput (QPS) or latency under different recall targets and top-k values.

Default setting. Unless otherwise specified, all experiments use the following default: HNSW [38] with SIFT-10M, QPS at top-k =100 and recall=0.9, with parameter search space `M` at an interval of 4 from [4, 64] and `EFC` at an interval of 4 from [4, 64]. The same applies to `R` and `L` in NSG.

4.2 End-to-End Evaluation

HNSW evaluation. Table 1 compares 14 tuning settings spanning two datasets, two metrics, three top-k values, and four recall targets. Compared to Grid Search, GArena w/o Pruning consistently finds the optimal configuration while reducing tuning time by 72–95% (e.g., from 3.2–3.7 hours down to 10–30 minutes). The major source of this speedup is the elimination of redundant distance computations (Figure 14). GArena w/ Pruning further reduces tuning time by 79–97% compared to Grid Search. As shown in Table 1 (indicated by red arrows), this variant occasionally incurs a minor acceptable quality degradation (<5%). This occurs because the pre-screening is inherently heuristic.

We also compared GArena against VDTuner under equal wall-clock time budgets (Budget 1 matches GArena w/o Pruning's time; Budget 2 matches GArena w/ Pruning's time). VDTuner's selected configuration can underperform the optimal index by 28% (Budget 2), highlighting the fragility of learning-based tuners when faced with the high per-trial cost of GANNS construction. Figure 12 further visualizes the best QPS achieved so far against cumulative tuning time. Benefiting from its rapid construction capability, GArena achieves convergence to the optimal performance in significantly less cumulative time than other competitors.

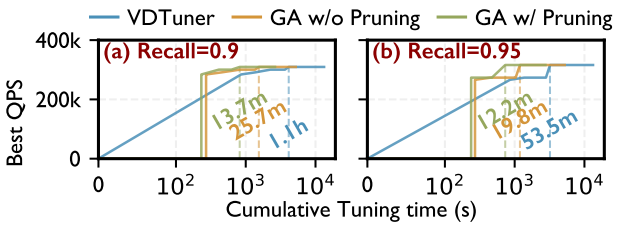

Figure 12: Best searched performance vs. cumulative tuning time. *GA denotes GArena; its curves start at zero because it evaluates indexes in batches.*

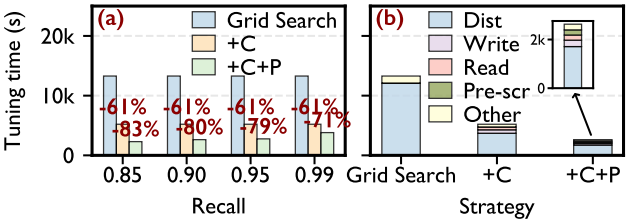

Figure 13: Ablation study on component contributions. *(a) Tuning time comparison. (b) Time breakdown analysis for recall=0.9. C: interval-based distance cache; P: performance-potential-guided pre-screening; Pre-scr: pre-screening.*

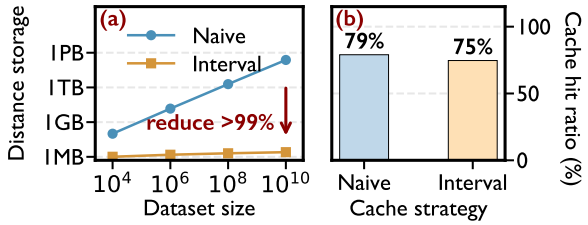

Figure 14: Interval-based distance cache effectiveness analysis. *(a) Cache memory footprints. (b) Cache hit ratios.*

NSG evaluation. Table 2 reports results on NSG. GArena w/o Pruning keeps the parameter optimality while reducing tuning time by 89–93% (from 7.8–8.2 hours down to 34–52 minutes). GArena w/ Pruning further reduces tuning time by 93–97% compared to Grid Search. Under equal time budgets, GArena produces graph indices that deliver up to 1.8x higher QPS compared to VDTuner. Compared with HNSW, this more pronounced result is due to the significantly higher distance computation redundancy (~92%) (Figure 3). Consequently, GArena's interval-based distance cache captures an exceptionally high hit rate during parallel construction, eliminating an even larger fraction of the computational bottleneck.

4.3 Technical Analysis

Ablation study. To understand the contribution of each component in GArena, we conduct ablation studies. Figure 13a shows the impact of adding the interval-based distance cache (C) and performance-potential-guided pre-screening (P) to Grid Search step by step. The distance cache alone reduces

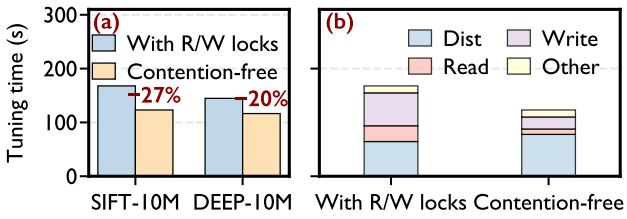

Figure 15: Comparison of a naive cache with fine-grained R/W locks and our contention-free cache. *(a) Tuning time across datasets. (b) Time breakdown on SIFT-10M.*

tuning time by 61% compared to Grid Search, while further adding pre-screening reduces time by 71–83% compared to Grid Search. Figure 13b provides a detailed breakdown analysis for recall=0.9. Compared to Grid Search, adding the cache (+C) drastically alleviates the computational bottleneck by eliminating 70% of high-dimensional distance calculations. While maintaining the cache inherently introduces new operations, the combined overhead of reading and writing the cache accounts for only 19%. This demonstrates the effectiveness of our cache in bypassing redundant distance computations.

By further adding pre-screening (+P), The pre-screening phase itself is highly lightweight, accounting for only 8% of the total tuning time, yet it further reduces the overall time by 49.5%. Although pre-screening prunes 50% of configurations, the time reduction is not exactly 50%, as complex, time-consuming configurations often exhibit higher potential and survive the pruning.

Interval-based distance cache. Figure 14 evaluates the effectiveness of interval-based distance cache by comparing it with the naive global cache (stated in §3.1.1). Figure 14a illustrates the memory footprint of both caches across different dataset scales. Naive distance cache grows super-linearly with dataset size and eventually hits the memory wall, while interval-based cache maintains a strictly bounded storage limit. Figure 14b further shows hit ratios of two caches. Although our interval-based cache is flushed at the boundary of each interval, its overall hit ratio is only 4% lower than that of the naive global cache. This demonstrates that our method successfully controls the memory overhead while simultaneously achieving efficient deduplication of distance computations.

Contention-free compute-ahead pipeline. We compare our single-writer/multi-reader cache architecture with a compute-ahead pipeline against a baseline denoted "with R/W locks", where every configuration concurrently reads from and writes to the global distance cache protected by fine-grained read/write locks. As shown in Figure 15a, our contention-free design reduces tuning time by 20–27% across datasets. Figure 15b breaks down the tuning time: although the contention-free cache increases distance computation by 13s due to a 6% lower hit ratio, it reduces cache read/write time by 58s, yielding a 27% net reduction in total tuning time, confirming

Metric	top-k	Recall	SP	PP	SP+PP
QPS	100	0.9	299.7k ↓	299.7k ↓	309.5k
QPS	10	0.9	386.6k ↓	385.9k ↓	413.5k
QPS	1000	0.9	25.0k ↓	22.1k ↓	25.7k
QPS	100	0.85	326.7k ↓	332.2k ↓	355.7k
QPS	100	0.95	310.8k ↓	310.8k ↓	316.1k
QPS	100	0.99	248.4k	248.4k	248.4k
Latency (ms)	100	0.9	17.3	17.3	17.3

Table 3: Comparison of scoring strategies for pre-screening. *SP: subset performance, PP: Performance Potential, SP+PP: combined score. Red ↓: cause best-found performance degradation.*

System	Time (h)	Time Reduction
Grid Search	70.9	—
VDTuner	54.3	23.4%
GArena	13.0	81.7%

Table 4: Tuning time on SIFT-1B.

that eliminating synchronization overhead far outweighs the marginal loss in cache hit ratio.

Performance-potential-guided pre-screening. We further evaluate different scoring strategies for pre-screening. As shown in Table 3, relying on a single metric significantly increases the risk of erroneously pruning optimal configurations, causing up to 14% performance degradation. Our combined SP+PP score consistently achieves the best performance.

4.4 Results on Billion-scale Datasets

To demonstrate GArena's scalability, we compare GArena with Grid Search and VDTuner on the billion-scale SIFT-1B dataset with 16 configurations ($\texttt{M}\in\{20,24,28,32\}$, $\texttt{EFC}\in\{20,24,28,32\}$). Table 4 shows the tuning time comparison. Grid Search exhaustively evaluates all 16 configurations, requiring approximately 2 days and 23 hours. VDTuner reduces this to 2 days and 6 hours (23.4% reduction) via Bayesian optimization. GArena completes the search in only 13.0 hours—an 81.7% time reduction over Grid Search and 76.1% over VDTuner. This demonstrates that GArena maintains substantial efficiency advantages even at billion-scale, making practical parameter tuning feasible for large-scale GANNS.

5 Related Work

Vector index parameter tuning. Parameter tuning [39, 40] aims to identify the optimal one via massive exploration. Existing tuning paradigms generally fall into three categories: heuristic-based (random/grid search [39]), and learning-based (e.g., BO [41, 42] or reinforcement learning [43]). Based on these methods, domain-specific tuning has flourished, including database systems [44], neural architecture search [45], and tensor program compilation [46, 47].

For vector index parameter tuning, it can be categorized into runtime tuning and construction-time tuning. Runtime tuning [48–50] adjusts query-dependent parameters without index reconstruction. Typical methods employ learned models such as neural networks [48] and GBDT [35, 49] to predict optimal query parameters. The abundance of queries and their short duration enable these methods to gather training data efficiently, leading to high-quality parameter selection. Conversely, construction-time tuning is notoriously challenging because evaluating the index quality requires an end-to-end construction from scratch. SPANN [51] and VDTuner [28] apply BO (via either custom implementations or general-purpose toolkit like NNI [31]) to search for construction parameters. These methods face the data scarcity issue as stated in §2. VSAG [13] proposed a labeling-based approach to encapsulate multiple graph degrees within a single physical graph, theoretically avoiding rebuilding. However, this method is restricted to tuning only the graph degree and inherently alters the semantics of the original graph algorithm, inevitably compromising search accuracy.

Vector index construction acceleration. A parallel line of research [17, 52, 53] focuses on accelerating the vector index construction process itself (a single trial in GArena). LSH-APG [17] uses locality-sensitive hashing to fast-track approximate proximity graph building. PiPNN [53] mitigates the search bottleneck in each insertion using a history-independent hash-based pruning algorithm. These approaches achieve acceleration by modifying the underlying graph construction algorithms, which alters the semantics or search properties of the original proximity graphs. They are orthogonal to GArena and users can opt-in to integrate them into GArena to further accelerate construction.

6 Conclusion

Graph-based ANNS algorithms require careful parameter tuning for unlocking optimal performance, but the high cost of index construction makes this process prohibitively expensive. We identify that over 70% of distance calculations are redundant across different parameter configurations. We propose GArena, which combines an interval-based distance cache for cross-configuration computation reuse and a performance potential predictor for small-scale pre-screening. Experimental results demonstrate that GArena achieves orders of magnitude speedup in tuning time while robustly finding the parameter-optimal index.

References

[1] Christian Platzer and Schahram Dustdar. A vector space search engine for web services. In *Third European Conference on Web Services (ECOWS'05)*, pages 9–pp. IEEE, 2005.

[2] Vaishali S Vairale and Samiksha Shukla. Recommendation of food items for thyroid patients using content-based knn method. In *Data Science and Security: Proceedings of IDSCS 2020*, pages 71–77. Springer, 2021.

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

[4] Meng Chen, Kai Zhang, Zhenying He, Yinan Jing, and X Sean Wang. Roargraph: A projected bipartite graph for efficient cross-modal approximate nearest neighbor search. *arXiv preprint arXiv:2408.08933*, 2024.

[5] Cong Fu, Changxu Wang, and Deng Cai. High dimensional similarity search with satellite system graph: Efficiency, scalability, and unindexed query compatibility. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(8):4139–4150, 2021.

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

[7] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering*, 32(8):1475–1488, 2019.

[8] Yun Peng, Byron Choi, Tsz Nam Chan, Jianye Yang, and Jianliang Xu. Efficient approximate nearest neighbor search in multi-dimensional databases. *Proceedings of the ACM on Management of Data*, 1(1):1–27, 2023.

[9] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *arXiv preprint arXiv:2101.12631*, 2021.

[10] Haodi Jiang, Hao Guo, Minhui Xie, Jiwu Shu, and Youyou Lu. High-throughput, cost-effective billion-scale vector search with a single gpu. *Proceedings of the ACM on Management of Data*, 3(6):1–27, 2025.

[11] Hao Guo and Youyou Lu. Achieving {Low-Latency}{Graph-Based} vector search via aligning {Best-First} search algorithm with {SSD}. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 171–186, 2025.

[12] Benjamin Coleman, Santiago Segarra, Alexander J Smola, and Anshumali Shrivastava. Graph reordering for cache-efficient near neighbor search. *Advances in Neural Information Processing Systems*, 35:38488–38500, 2022.

[13] Xiaoyao Zhong, Haotian Li, Jiabao Jin, Mingyu Yang, Deming Chu, Xiangyu Wang, Zhitao Shen, Wei Jia, George Gu, Yi Xie, et al. Vsag: An optimized search framework for graph-based approximate nearest neighbor search. *arXiv preprint arXiv:2503.17911*, 2025.

[14] Javier A Vargas Munoz, Zanoni Dias, and Ricardo da S. Torres. A genetic programming approach for searching on nearest neighbors graphs. In *Proceedings of the 2019 on International Conference on Multimedia Retrieval*, pages 43–47, 2019.

[15] Xiaoliang Xu, Mengzhao Wang, Yuxiang Wang, and Dingcheng Ma. Two-stage routing with optimized guided search and greedy algorithm on proximity graph. *Knowledge-Based Systems*, 229:107305, 2021.

[16] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. Freshdiskann: A fast and accurate graph-based ann index for streaming similarity search. *arXiv preprint arXiv:2105.09613*, 2021.

[17] Xi Zhao, Yao Tian, Kai Huang, Bolong Zheng, and Xiaofang Zhou. Towards efficient index construction and approximate nearest neighbor search in high-dimensional spaces. *Proceedings of the VLDB Endowment*, 16(8):1979–1991, 2023.

[18] Shuo Yang, Jiadong Xie, Yingfan Liu, and Jeffrey Xu Yu. Revisiting the index construction of proximity graph-based approximate nearest neighbor search. *Proceedings of the ACM on Management of Data*, 2(1):1–25, 2024.

[19] Naoki Ono and Yusuke Matsui. Relative nn-descent: A fast index construction for graph-based approximate nearest neighbor search. In *Proceedings of the 31st ACM International Conference on Multimedia*, pages 1659–1667, 2023.

[20] Zhonggen Li, Xiangyu Ke, Yifan Zhu, Bocheng Yu, Baihua Zheng, and Yunjun Gao. Scalable graph indexing using gpus for approximate nearest neighbor search. *arXiv preprint arXiv:2508.08744*, 2025.

[21] Hiroyuki Ootomo, Akira Naruse, Corey Nolet, Ray Wang, Tamas Feher, and Yong Wang. Cagra: Highly parallel graph construction and approximate nearest neighbor search for gpus. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 4236–4247. IEEE, 2024.

[22] Yicheng Jin, Yongji Wu, Wenjun Hu, and Bruce M Maggs. Curator: Efficient indexing for multi-tenant vector databases. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 345–362, 2024.

[23] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, et al. Spfresh: Incremental in-place update for billion-scale vector search. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 545–561, 2023.

[24] Yury A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 40(9):1996–2012, 2018.

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

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

[27] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *arXiv preprint arXiv:1707.00143*, 2017.

[28] Tiannuo Yang, Wen Hu, Wangqi Peng, Yusen Li, Jianguo Li, Gang Wang, and Xiaoguang Liu. Vdtuner: Automated performance tuning for vector data management systems. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 4357–4369. IEEE, 2024.

[29] Microsoft. Sptag auto-tune tool. https://github.com/microsoft/SPTAG/blob/main/Tools/nni-auto-tune/README.md, 2026.

[30] Microsoft. Sptag/tools/nni-auto-tune/config.yml at main · microsoft/sptag · github. https://github.com/microsoft/SPTAG/blob/main/Tools/nni-auto-tune/config.yml, 2026.

[31] Microsoft. Neural network intelligence (nni). https://github.com/microsoft/nni, 2026.

[32] Mengzhao Wang, Haotian Wu, Xiangyu Ke, Yunjun Gao, Yifan Zhu, and Wenchao Zhou. Accelerating graph indexing for anns on modern cpus. *Proceedings of the ACM on Management of Data*, 3(3):1–29, 2025.

[33] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. Pearson correlation coefficient. In *Noise reduction in speech processing*, pages 1–4. Springer, 2009.

[34] James William Thomas. *Numerical partial differential equations: finite difference methods*, volume 22. Springer Science & Business Media, 2013.

[35] Rentong Guo, Xiaofan Luan, Long Xiang, Xiao Yan, Xiaomeng Yi, Jigao Luo, Qianya Cheng, Weizhi Xu, Jiarui Luo, Frank Liu, et al. Manu: a cloud native vector database management system. *arXiv preprint arXiv:2206.13843*, 2022.

[36] Big ANN Benchmarks. BIG-ANN benchmarks: Neurips'21 track – dataset details. Web page, 2021.

[37] Yandex Research. Benchmarks for billion-scale similarity search. Blog post, 2019.

[38] FAISS Contributors. FAISS: A library for efficient similarity search and clustering of dense vectors. GitHub repository.

[39] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *The journal of machine learning research*, 13(1):281–305, 2012.

[40] Luca Franceschi, Michele Donini, Valerio Perrone, Aaron Klein, Cédric Archambeau, Matthias Seeger, Massimiliano Pontil, and Paolo Frasconi. Hyperparameter optimization in machine learning. *arXiv preprint arXiv:2410.22854*, 2024.

[41] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25, 2012.

[42] Peter I Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

[43] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2017.

[44] Xinyang Zhao, Xuanhe Zhou, and Guoliang Li. Automatic database knob tuning: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 35(12):12470–12490, 2023.

[45] Quanlu Zhang, Zhenhua Han, Fan Yang, Yuge Zhang, Zhe Liu, Mao Yang, and Lidong Zhou. Retiarii: A deep learning {Exploratory-Training} framework. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 919–936, 2020.

[46] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 863–879, 2020.

[47] Isu Jeong and Seulki Lee. Bayesian code diffusion for efficient automatic deep learning program optimization. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 295–311, 2025.

[48] Pengcheng Zhang, Bin Yao, Chao Gao, Bin Wu, Xiao He, Feifei Li, Yuanfei Lu, Chaoqun Zhan, and Feilong Tang. Learning-based query optimization for multi-probe approximate nearest neighbor search: P. zhang et al. *The VLDB Journal*, 32(3):623–645, 2023.

[49] Oleg Senkevich, Siyang Xu, Tianyi Jiang, Alexander Radionov, Jan Tabaszewski, Dmitriy Malyshev, Zijian Li, Daihao Xue, Licheng Yu, Weidi Zeng, et al. Kscann: Scalable approximate nearest neighbor search on kunpeng. *arXiv preprint arXiv:2511.03298*, 2025.

[50] Zehai Yang and Shimin Chen. Rairs: Optimizing redundant assignment and list layout for ivf-based ann search. *arXiv preprint arXiv:2601.07183*, 2026.

[51] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. Spann: Highly-efficient billion-scale approximate nearest neighborhood search. *Advances in Neural Information Processing Systems*, 34:5199–5212, 2021.

[52] Yang Shi, Yiping Sun, Jiaolong Du, Xiaocheng Zhong, Zhiyong Wang, and Yao Hu. Scalable overload-aware graph-based index construction for 10-billion-scale vector similarity search. In *Companion Proceedings of the ACM on Web Conference 2025*, pages 1303–1307, 2025.

[53] Tobias Rubel, Richard Wen, Laxman Dhulipala, Lars Gottesbüren, Rajesh Jayaram, and Jakub Łącki. Pipnn: Ultra-scalable graph-based nearest neighbor indexing. *arXiv preprint arXiv:2602.21247*, 2026.