Scaling Laws for Nearest Neighbor Search

Philip Sun [1] **Felix Chern** [1] **Yaroslav Akhremtsev** [1] **Ruiqi Guo** [1] **David Simcha** [1] **Sanjiv Kumar** [1]

Abstract

This paper investigates the scaling laws that characterize the performance of various nearest neighbor search algorithms across a number of operational scenarios. We analyze the asymptotic costs of indexing, storage, and compute for the three dominant paradigms in nearest neighbor search algorithms: brute-force, partitioning-based, and graph-based. We find that these three families of algorithms make fundamentally different tradeoffs between the three costs, leading to each algorithm having its own advantages and disadvantages. Our work challenges prior notions of a single "best" nearest neighbor search algorithm, instead suggesting the optimum is setup-dependent.

1. Introduction

Nearest neighbor search is a foundational problem in computer science and machine learning, concerned with finding the points in a given dataset $D \in \mathbb{R}^{N \times d}$ that are closest to a query $q \in \mathbb{R}^d$. Its applications include retrieval-augmented generation and recommender systems, among many others. One important subproblem within this field is approximate nearest neighbor (ANN) search algorithms, which sacrifice some loss in accuracy for a potentially significant increase in search throughput and/or reduction in latency.

ANN algorithm research is concerned with minimizing the costs associated with handling nearest neighbor queries at a given accuracy. This, however, is difficult to quantify because there is a heterogeneous mix of costs: before any ANN queries can be handled, an index must first be created, often at non-trivial expense, leading to an indexing cost (I). And at query time, answering the queries themselves incurs computational cost (C), but additionally the index itself must be persisted in some storage medium, leading to a fixed storage cost (S) that must be paid regardless of the query frequency. We therefore have three cost components, two of which (indexing and querying) must be multiplied by their respective frequency of occurrence, which we denote f_I and f_C. This results in a total cost of $f_I \cdot I + f_C \cdot C + S$.

Table 1. Asymptotic cost of the three components of nearest neighbor search cost with respect to dataset size, N.

Algorithm	Indexing Cost (I)	Compute Cost (C)	Storage Cost (S)
Brute Force	$O(N)$	$O(N)$	$O(N)$
Partitioning-Based	$\omega(N)$	$o(N)$	$O(N)$
Graph (1-Machine)	$\omega(N)$	$o(N)$	$O(N)$
Graph (Distributed)	$O(N)$	$O(N)$	$O(N)$

Optimizing ANN index efficiency is therefore complex, involving tradeoffs across these components, and also implicitly dependent on f_I and f_C. Moreover, these three components are not static, but rather dependent upon the dataset size, N. ANN cost J for a given algorithm can therefore be defined as the following, with respect to N:

$$J(N) = f_I \cdot I(N) + f_C \cdot C(N) + S(N). \quad (1)$$

The field of ANN algorithms has a substantial history with numerous proposed techniques, but past works have primarily focused on the comparison and improvement of query-time compute cost C while neglecting analysis of the other cost components I and S, which are often significant and sometimes even dominant. Additionally, relatively little work has been done quantifying how modern ANN algorithms scale with respect to N; even algorithms with abundant theoretical analysis, e.g. LSH and k-d tree, rarely can predict search cost or recall on modern datasets and hardware with substantial accuracy.

This paper presents an analysis of these scaling laws for nearest neighbor search. Our primary contributions are:

- We systematically analyze the asymptotic costs of indexing, storage, and query computation for three dominant nearest neighbor search paradigms: brute-force, partitioning-based, and graph-based algorithms. Our analysis (summarized in Table 1) shows fundamental tradeoffs these families make, resulting in different optimal use cases.

[1]Google Research, New York, USA. Correspondence to: Philip Sun <sunphil@google.com>.

Proceedings of the 1st Workshop on Vector Databases at International Conference on Machine Learning, 2025.

- Through empirical investigation, we observe a consistent power-law query cost scaling relationship for ANN algorithms, where query compute $C(N)$ scales proportionally to N^α with $\alpha < 1$. This furthermore implies indexing costs generally scale as $O(N^{1+\alpha})$ due to ANN indexing's current recursive dependence on nearest neighbor search itself.

- We re-evaluate brute-force search, demonstrating that with modern hardware accelerators (e.g., Tensor Processing Units), its high arithmetic intensity and regular memory access can lead to competitive throughput and total cost of ownership, particularly for datasets with millions or fewer vectors, avoiding the complex and often superlinear indexing costs of approximate methods.

- Finally, we investigate the impact of distributed environments on ANN scaling, necessary for datasets exceeding single-machine capacity. We show that distributing via random sharding increases query cost to $O(N)$. Meanwhile, distributing via a globally trained index can maintain more favorable sublinear scaling, but incurs network communication overhead. This overhead is significant for graph-based algorithms and explains why thus far, only partitioning-based algorithms have achieved sublinear scaling for distributed-scale ANN datasets.

By clarifying these scaling behaviors, this work suggests that there is no single "best" nearest neighbor search algorithm. Instead, we argue that the optimal choice depends on the specific operational scenario, including dataset characteristics, query load, latency requirements, hardware availability, and cost constraints. We expect these findings will guide practitioners in making informed architectural decisions and encourage further theoretical work and empirical evaluation.

2. Preliminaries

This section formally defines the nearest neighbor search problem, the metrics used to evaluate approximate solutions, and introduces the core algorithmic paradigms discussed in this work.

2.1. Nearest Neighbor Search

Let $D = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\}$ be a dataset of N data points, where each point $\mathbf{x}_i \in \mathbb{R}^d$ is a d-dimensional vector. Given a query $q \in \mathbb{R}^d$ and a distance metric $\delta : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}_{\geq 0}$ or a similarity score that can be transformed into a metric (e.g., Euclidean distance $\delta(\mathbf{a}, \mathbf{b}) = \|\mathbf{a} - \mathbf{b}\|_2$, or cosine similarity), the **Nearest Neighbor Search** problem is to find a point $\mathbf{x}^* \in D$ such that:

$$\delta(\mathbf{q}, \mathbf{x}^*) \leq \delta(\mathbf{q}, \mathbf{x}_i) \quad \forall \mathbf{x}_i \in D.$$

This also generalizes to finding the k nearest neighbors that minimize δ, which is commonly used in practice and will be the focus of our remaining analysis.

2.2. Approximate Nearest Neighbor Search and Recall

For large datasets or in high dimensions, finding the exact nearest neighbors can be computationally prohibitive. **Approximate Nearest Neighbor (ANN)** search algorithms aim to find points that are "close enough" to the true nearest neighbors with significantly reduced computational cost.

A primary metric to evaluate the quality of ANN algorithms is **recall**. For a given query $\mathbf{q}$ and a desired number of neighbors k, let $T_k(\mathbf{q})$ be the set of true k nearest neighbors and $A_k(\mathbf{q})$ be the set of k neighbors returned by an ANN algorithm. The recall for this query is defined as:

$$\text{Recall@}k(\mathbf{q}) = \frac{|T_k(\mathbf{q}) \cap A_k(\mathbf{q})|}{k}$$

In practice, recall is often averaged over a representative set of queries. Many applications target a specific recall level (e.g., 90% or 99% for $k = 1$) as a quality threshold.

2.3. Query Cost $C(N)$

We define $C(N)$ as the *query-time compute cost* for a dataset of size N. This represents the computational resources required to process a single query and return its approximate nearest neighbors at a pre-defined target recall. $C(N)$ implicitly depends on the dimensionality d, the chosen algorithm, its specific hyperparameters, and the underlying hardware. Throughout this paper, when discussing $C(N)$, we assume these other factors are held constant or scaled appropriately as N changes, allowing us to focus on the relationship between query cost and dataset size.

2.4. Dominant ANN Paradigms

The vast landscape of ANN algorithms can be broadly categorized. We focus on three dominant paradigms whose scaling properties are central to this work:

- **Brute-Force (Linear Scan)**: This straightforward approach computes the distance from the query $\mathbf{q}$ to every point $\mathbf{x}_i \in D$ and returns the points with the smallest distances. While guaranteeing perfect recall, its query cost $C(N)$ scales linearly with N, i.e., $C(N) = \Theta(Nd)$. Its simplicity and amenability to hardware acceleration are key characteristics.

- **Partitioning-Based Methods**: These algorithms build an index by recursively dividing the vector space (e.g.,

k-d trees) or the dataset itself (e.g., k-means clustering) into smaller regions or clusters. At query time, only a subset of these regions—those likely to contain the nearest neighbor—are explored. This pruning aims to achieve sub-linear query cost $C(N) = o(N)$.

- **Graph-Based Methods**: These methods construct a proximity graph where data points are nodes, and an edge exists between two nodes if they are considered "close" under some heuristic (e.g., HNSW, NSG). Queries are answered by traversing this graph, typically starting from an entry point and greedily moving towards nodes closer to the query, also striving for $C(N) = o(N)$.

The specific mechanisms by which these paradigms achieve their respective cost-recall tradeoffs, and how these mechanisms influence their scaling with N, form the core investigation of this paper.

3. Related Work

3.1. Theoretical Guarantees for ANN Algorithms

3.1.1. LSH

Locality-sensitive hashing (LSH) techniques for ANN have very thorough theoretical analyses, but their results are typically done for the c-approximate nearest neighbors problem, where the goal is to return any vector whose distance is within a multiplicative factor of c of the true closest vector (Andoni et al., 2015). When a typical 90% recall target is translated into a c-approximation ratio, the resulting c is usually very close to 1, making LSH degenerate into an $O(N^{1-\epsilon})$ algorithm roughly equivalent to linear scan, but with much higher memory usage and constant factor overhead. Indeed, LSH-style ANN libraries have not demonstrated competitive empirical results on modern datasets relative to graph-based or partitioning-based algorithms.

3.1.2. k-D Tree Variants

A number of k-d tree variants have also been developed for the ANN problem, using techniques such as randomization (Dasgupta & Freund, 2008) and overlapping cells (Dasgupta & Sinha, 2013) to improve performance over the standard k-d tree. These algorithms have rigorous upper bounds for their time complexity and probability of failing to find the true nearest neighbor, but the failure probability bound is typically trivial (near 1) on real-world datasets.

3.1.3. Graph Algorithms

With regards to graph-based ANN algorithms, the Delaunay triangulation of the datapoints is frequently leveraged in theoretical analyses, because as discussed in (Navarro, 1999), greedily walking along the Delaunay triangulation allows the nearest neighbor for any query vector to be found, with no backtracking needed. However, the Delaunay triangulation may be nearly fully-connected, making graph search equivalent to brute-force. The paper (Dwyer, 1989), notably cited in the HNSW analysis (Malkov & Yashunin, 2020), shows that the Delaunay triangulation of points uniformly sampled from a d-dimensional ball has constant expected average degree. This constant, however, is extremely loosely bounded and greater than 10^{13} for $d = 16$, making such analysis serve little practical use.

Sparsifying the Delaunay graph makes the graph search procedure less-obviously brute force in nature, but also greatly weakens guarantees that the search procedure will find the true nearest neighbor. For example, the sparse neighborhood graph (Arya & Mount, 1993) has much stricter conditions for creating an edge, leading to lower average vertex degree, but also only guarantees finding a query q's nearest neighbor if q was in the indexed dataset already. Note this is a trivial guarantee solvable with a hash map and no specialized nearest neighbor data structure at all. Even for this graph, the degree bound is weak, and exponential with respect to dimensionality.

While graph-based ANN indices have empirically shown their ability to produce high-performing and high-recall results, current theory cannot fully explain their success.

3.2. LLM Scaling Laws

Our work is more similar in nature to the literature in LLM scaling laws. The foundational paper in this field was written by Kaplan et al. (2020) and found power laws relating model size, dataset size, and training-time compute with the cross-entropy language modeling loss. Further works in this field include (Hoffmann et al., 2022), which highlighted the importance of training with more data, and (Muennighoff et al., 2025), which tried to extrapolate an analogous scaling law for increased inference-time compute. In a different vein, (Sharma & Kaplan, 2020) tried to derive a theoretical justification for the observed scaling laws. We hope that, similar to the works in LLM scaling laws, our observations can improve ANN efficiency by better allocating hardware resources to match the specific demands of the ANN workload, while also laying the groundwork for more theoretical justifications of these results in the future.

4. Querying Cost Scaling

From empirical observations, we find a trend between dataset size and query-time compute cost that is remarkably consistent over a range of ANN algorithms and three orders of magnitude of dataset size; when targeting 90%

Recall@10, we find that

$$C(N) \propto N^{1/3}. \tag{2}$$

The groundwork for our observation is shown in Figure 1, where we perform a log-log linear regression between N and ANN throughput per unit of computational power. Across all three algorithms evaluated, we find a fairly consistent slope of approximately $-1/3$, which implies

$$\log\left(\frac{k_1}{C(N)}\right) \approx -\frac{1}{3}\log N + k_2$$

for some constants k_1 and k_2, which after exponentiation lead to our result in equation 2.

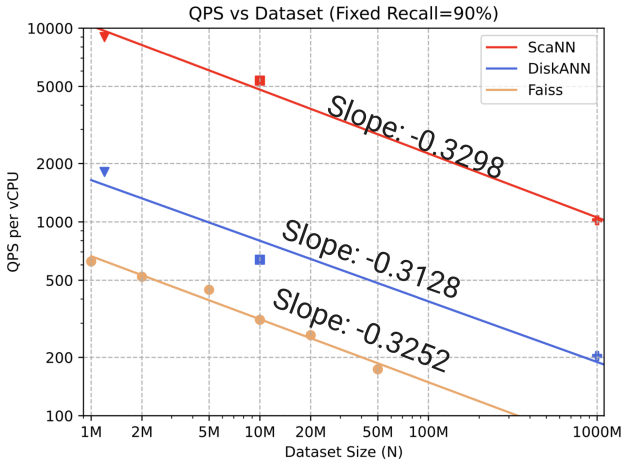

Figure 1. Across three orders of magnitude of dataset size and three ANN algorithm implementations, we see the $N^{1/3}$ relationship holding with quite strong consistency.

We caveat equation 2 with the fact that computing the exact trend of nearest neighbor search cost with respect to dataset size is difficult due to the number of hyperparameters most ANN algorithms have. It is particularly difficult to determine how indexing time should be scaled with dataset size (linear, quadratic, or otherwise); allocating relatively greater amounts of indexing compute budget towards larger datasets would give the appearance that $C(N)$ grows more slowly.

An additional difficulty encountered in computing this trend is dataset distribution differences. The ScaNN (Guo et al., 2020; Sun et al., 2023) and DiskANN (Subramanya et al., 2019) datapoints were taken from three different datasets (see Appendix A.1), so differences in intrinsic dataset dimensionality contribute to performance differences in addition to the effect of scaling N, adding noise to our analysis. The datasets used all had dimensionality between 100 and 200, and the cube root law is likely particular to this range. Other datasets with greater intrinsic dimensionality will require more search effort to achieve the same recall, resulting in similar scaling to higher recall targets (discussed in the following Section 4.1) on the analyzed datasets.

4.1. Generalization to Additional Recall Targets

The $N^{1/3}$ scaling seems particular to the 90% Recall@10 target, and we expect this to generalize to a power law for other recall targets, i.e. $C(N) \propto N^{\alpha}$ for some α. Indeed, (Douze et al., 2025) observed $\alpha = 0.29$ and $\alpha = 0.45$ for the 50% and 99% recall targets, respectively. At the limit, we expect $\alpha \to 0$ as the recall target approaches zero, because returning an arbitrary k elements in $O(1)$ time satisfies the zero-recall target. Conversely, we expect $\alpha \to 1$ as the recall target approaches one, where only linear scan can guarantee exact results. Interpolating a curve to exactly model the relationship between recall target and α is a promising direction for future work.

4.2. $C(N)$: Further Breakdown and Analysis

We can decompose $C(N)$ as follows:

$$C(N) = c_0 \cdot Nd \cdot f(N) \tag{3}$$

where c_0 is the computational cost associated with accessing one byte of the ANN index and $f(N)$ is the filtration ratio, equal to the proportion of the entire ANN index that must be accessed in order to answer a query with a given expected recall. In essence, $Nd \cdot f(N)$ is the amount of memory read to answer the query, and c_0 is the cost per read. The choice of a constant c_0 independent of dataset size can be justified by noting most ANN index data will fit on one uniform storage medium and disregarding caching effects; c_0 is determined by the raw performance of the storage medium itself, and the read access pattern (granularity, predictability) determined by the algorithm. Rearranging our terms we find that

$$f(N) \propto N^{-2/3}$$

with the intuitive interpretation that the larger the dataset, the more effectively an ANN algorithm can prune the search space.

4.3. Brute Force Scaling

The above analysis only applies to approximate algorithms that index and prune the search space; for linear scan algorithms:

1. $C(N) = \Theta(N)$
2. $f(N) = O(1)$
3. c_0 is typically much lower than that of other ANN algorithms, due to contiguous memory access and amenability to SIMD acceleration. In the case of batched search, matrix multiplication hardware and high-bandwidth memory may be used, leading to extremely efficient implementation.

While (1) and (2) are well-known and fairly trivial algorithmic observations, we believe that the power of observation (3) may be underrecognized. This is highlighted in Figure 2, where we show nearest neighbor performance on a TPU (Chern et al., 2022) that provides $>$ 10x the QPS/TCO of a CPU-based ANN solution.

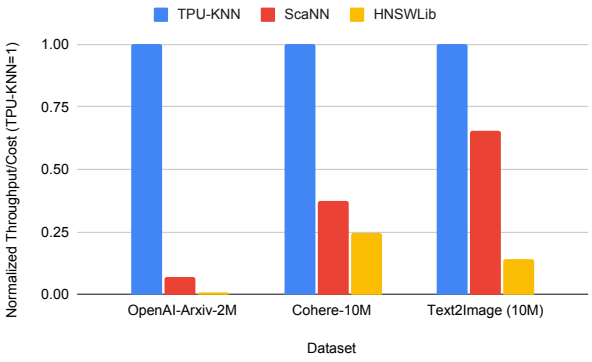

Figure 2. Brute force algorithms can uniquely leverage the extremely high bandwidth memory and high-throughput matrix multiplication units present on a TPU to significantly outperform more algorithmically sophisticated CPU techniques.

We can see that on millions-scale vector datasets, a size not uncommon for many real-world ANN deployments, accelerator-optimized linear scans can be the most TCO-optimal way to serve the ANN workload.

In addition to the cost advantage, linear scan has a number of other operational advantages: the algorithm has far fewer tuning knobs to configure, its recall is generally less data-dependent and therefore much more robust to distributional shifts or quirks, and indexing is trivial. The indexing cost advantage is discussed further in Section 6. The advantage of TPU-KNN is relatively smaller on the 10M vector datasets than the 2M vector datasets, which can likely be attributed to the sublinear $C(N)$ of ScaNN and HNSW providing a greater advantage for larger N. However, even at 10M scale, the TPU-based solution is the most cost-competitive.

5. Storage Cost Scaling

All practical nearest neighbor search algorithms have $\Theta(N)$ space complexity. More space is not needed because ANN algorithms only need a multiplicative constant overhead over the dataset itself, and from information theory, it is clear that arbitrary datasets cannot be compressed to occupy sublinear space without severe recall loss, because doing so would imply the compressibility of arbitrary information.

Among various $\Theta(N)$ approaches, however, constant factors differ significantly. First, the dataset's values can be stored in a number of binary formats; common choices include scalar-quantized 8-bit integers and 16, 32, and 64 bit floating point. For a dataset of N vectors and d dimensions, such encodings would lead to storage costs of Nd, $2Nd$, $4Nd$, and $8Nd$ bytes, respectively.

Next, ANN indices generally need to store additional index data besides the quantized dataset. Brute force indices have no such metadata, and their storage costs are exactly equal to the size of the underlying quantized dataset, making them a succinct data structure. Partitioning-based indices must store the partition centers and each datapoint's partition assignments, and frequently utilize further quantized datapoint representations to accelerate their search. This additional data is typically $10-30\%$ of the dataset's size. Graph-based indices typically have the highest storage overhead, because they require each datapoint to store its respective out-edges, sometimes more than doubling storage requirements relative to the dataset itself.

Finally, different storage technologies have widely varying costs. Examples of storage mediums include: bandwidth-optimized memory, such as HBM or GDDR6; standard CPU RAM, such as DDR5; and flash storage. Higher-bandwidth and lower-latency storage types enable greater ANN performance, but come at a greater cost, as shown in Table 2:

Table 2. Memory costs from Google Cloud Compute Engine

Product	Memory Type	Cost/GB /Month	Bandwidth /Cost (GB/s/$)
TPU v6e Chip	HBM	$61.59	0.832
c4-highmem-32	DDR5	$6.14	0.105
6TB NVMe SSD	Flash	$0.08	0.013

Optimizing S, the ANN index storage cost, is critical yet underacknowledged. Considering total ANN serving cost from Equation 1, for low query traffic volumes f_C, the storage cost can dominate the query-time compute cost: $S \gg f_C \cdot C$, so switching to cheaper storage can significantly decrease total serving cost $J(N)$. Conversely, for extremely high f_C, the proportion of cost coming from compute rises, and it becomes cost-advantageous to upgrade to faster storage to maximally utilize the computational resources (for example, moving from flash storage to RAM so that the CPU spends less time waiting on data, allowing fewer CPUs to be used).

6. Indexing Cost Scaling

ANN algorithms face a tradeoff between indexing speed and index quality: currently, algorithms cannot achieve sublinear query complexity without spending superlinear indexing time.

6.1. Brute Force Indexing

Brute-force ANN indices require minimal preparatory work, all of which can be done in linear time. Examples of necessary indexing operations include transposing between row-major and column-major storage formats, performing per-element quantization (for example, re-encoding 32-bit floats into the bfloat16 format), or copying data from host into accelerator memory.

6.2. Graph Indexing

Graph-based ANN algorithms typically create their index by incrementally inserting datapoints one by one. Each datapoint queries for the nearest neighbors among the already-inserted datapoints. These nearest neighbor results are used as the candidate set of vertices to connect to the current datapoint. The total computational complexity of these queries is $\sum_{x=1}^{N} C(x)$.

6.3. Partition-Based Indexing

The most expensive step in partition-based indexing is computing a k-means clustering of the dataset. This is typically done with Lloyd's algorithm, which alternates between assigning each datapoint to its closest center, and updating each center to equal the mean of its assignments. The latter step can be done in linear time and is not asymptotically significant in its costs. The former step is expensive; assuming the average cluster size is kept at a constant M, there are N/M cluster centers and N datapoints to assign to the clusters, leading to an assignment cost of $N \cdot C(N/M)$ per iteration of Lloyd's algorithm. A constant number of Lloyd's algorithms has empirically been sufficient for creating an effective ANN index, leading to a total computational complexity of $N \cdot C(N/M)$.

6.4. Asymptotic Analysis

Both graph-based and partitioning-based ANN algorithms run $O(N)$ nearest neighbor queries during indexing. Assuming, as discussed in Section 4, that $C(x) = O(x^{\alpha})$, this leads to both procedures having $O(N^{\alpha+1})$ indexing complexity:

$$\sum_{x=1}^{N} C(x) = O(N^{\alpha+1}), \qquad \text{(Graph-Based)}$$

$$N \cdot C(N/M) = O(N^{\alpha+1}). \qquad \text{(Partitioning-Based)}$$

Despite the equivalent asymptotic indexing costs of graph-based and partitioning-based ANN algorithms, the latter have generally demonstrated notably faster indexing performance in practice. This can be attributed to the fact that partition assignment in Lloyd's algorithm is trivially batchable and parallelizable. In contrast, graph-based indexing is sequential by nature and less amenable to efficient implementation, leading to a higher constant factor.

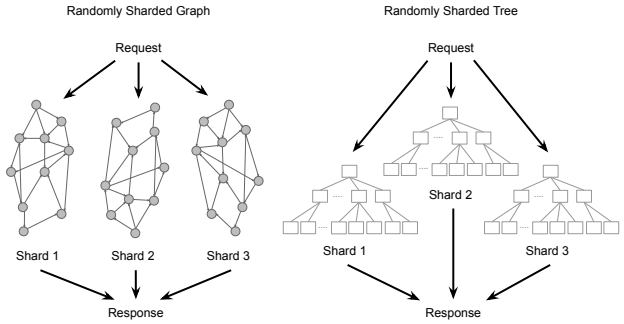

Figure 3. Random sharding for graph-based and partitioning-based ANN algorithms.

7. Distributed ANN and its Impact on Scaling

For sufficiently large datasets, a single machine no longer has the memory to serve the index by itself, and a distributed ANN setup is needed. Even disk-based ANN solutions are not exempt from this problem, because such solutions still typically require a significant portion of the index to be resident in RAM; flash memory has too coarse a read granularity and too little bandwidth to efficiently serve an entire ANN index without a faster storage medium for more frequently accessed data. For instance, DiskANN stores product-quantized vector representations in RAM, making its RAM footprint typically at least 10% of its disk footprint (Subramanya et al., 2019).

At a high level, there are two approaches to distributing an ANN index, each of which is discussed below.

7.1. Random Sharding

In random sharding, a dataset $X \in \mathbb{R}^{N \times d}$ is arbitrarily split into S shards of N/S datapoints each. This results in S datasets of size N/S; we independently construct an ANN index for every dataset, and during query time, we forward the query to every shard and then aggregate the final results together. A diagram of the query-time information flow is shown above, in Figure 3.

Critically, in random sharding, each shard works independently, and no inter-shard communication is necessary other than the final aggregation. This is in contrast to a globally partitioned setup, as discussed in Section 7.2.

7.1.1. Random Sharding: Asymptotic Analysis

Suppose that the maximum number of datapoints servable from a single machine is N_0. This constraint typically arises from a machine's memory capacity, and is therefore con-

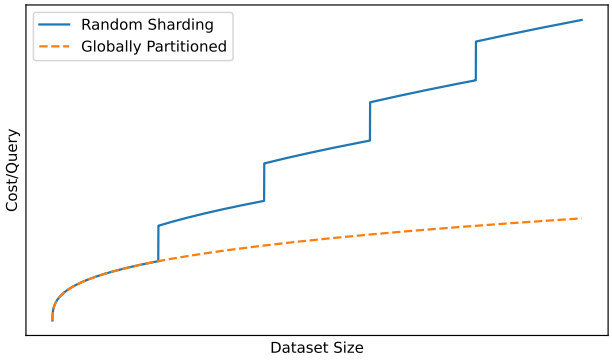

Figure 4. Random sharding leads to a discontinuity in search cost each time an additional shard is introduced. Asymptotically, this leads to linear scaling.

stant. The minimum number of shards necessary to serve a dataset of size N is therefore $S = \lceil N/N_0 \rceil$.

Intuitively, by bounding the shard size to N_0, we are also bounding the filtration ratio $f(N)$ from Equation 3 to a constant, preventing further efficiency gains from pruning further on larger indices. Formally, random sharding leads to linear asymptotic query cost:

$$S \cdot C(N/S) \leq S \cdot C(N_0) = \lceil N/N_0 \rceil \cdot O(1) = O(N).$$

This is illustrated in Figure 4. Indexing cost also becomes linear with respect to N:

$$S \cdot I(N/S) \leq S \cdot I(N_0) = \lceil N/N_0 \rceil \cdot O(1) = O(N).$$

Altogether, random sharding is asymptotically equivalent to linear scan, albeit with vastly different constant factors. For query-time compute C, random sharding and linear scan have constant factors of $c_0^{(a)} N_0^{\alpha}$ and $c_0^{(b)}$, respectively, where $c_0^{(a)}$ and $c_0^{(b)}$ denote the respective per-byte read costs for approximate and brute-force. For sufficiently large N_0, approximate algorithms may achieve a significant advantage. Conversely, the indexing time constant factors vastly favor linear-scan.

7.2. Globally Indexed

In global partitioning, a single unified index is created for the entire dataset, and then this index is sharded to fit in multiple machines. How this sharding is performed depends on the exact ANN algorithm used, which will be discussed below.

7.2.1. Global Partition-Based ANN

The k-means tree used in partitioning-based ANN algorithms is fairly straightforward to shard due to its relatively constrained structure. As shown in Figure 5, the upper levels of a recursive k-means tree are typically replicated across all shards, which reduces cross-shard communication overhead for a negligible increase in memory usage. The remaining levels of the tree are randomly sharded across the machines. Note that even though the partitions are randomly assigned to machines, the partitions themselves were first trained globally, distinguishing this scheme from the random sharding described in Section 7.2 and allowing the continuation of sublinear query cost scaling.

The query-time procedure for this index walks and prunes through the k-means tree in a breadth-first, beam search-like manner; at layer L, it finds the closest t_L partition centers and proceeds to evaluate the vectors contained within those partitions at layer $L + 1$. If this layer is sharded across machines rather than replicated, a round of network calls must be done to aggregate the global best partitions and then communicate the necessary work to be done at the next layer. This implies that $\log_M N$ network round trips are necessary if the average partition size is M; for a typical setup where M is in the hundreds, even trillions-scale datasets can be served with minimal communication overhead.

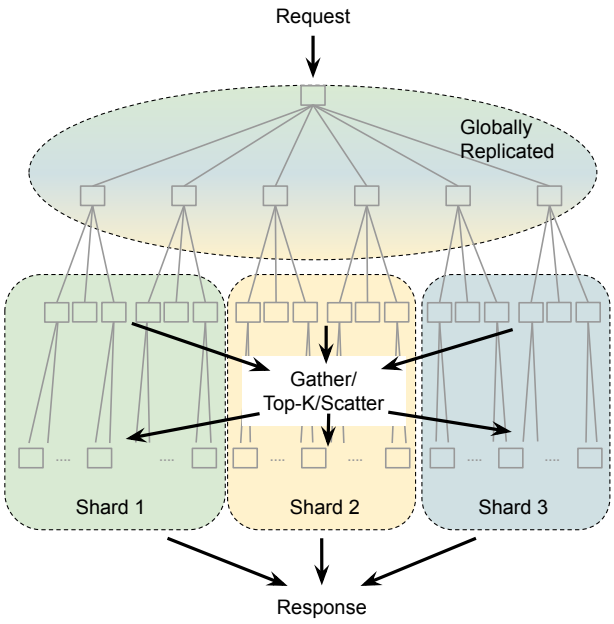

Figure 5. Partitioning-based ANN algorithms can be effectively sharded to maintain sublinear query cost even when dataset scale reaches a size requiring distributed serving.

7.2.2. Global Graph-Based ANN

Graphs, due to their inherently diminished structure relative to trees, are harder to partition and serve in a global index. The core problem is minimizing the number of inter-shard

("cut") edges between shards to minimize communication overhead, while keeping the partitions balanced in size. Examples of attempting to shard a graph, and the cut edges that result, are shown in Figure 6. Each cut edge is a point where a query's graph traversal must make a network hop to another machine. These network hops introduce significant latency, which can quickly dominate the overall query time.

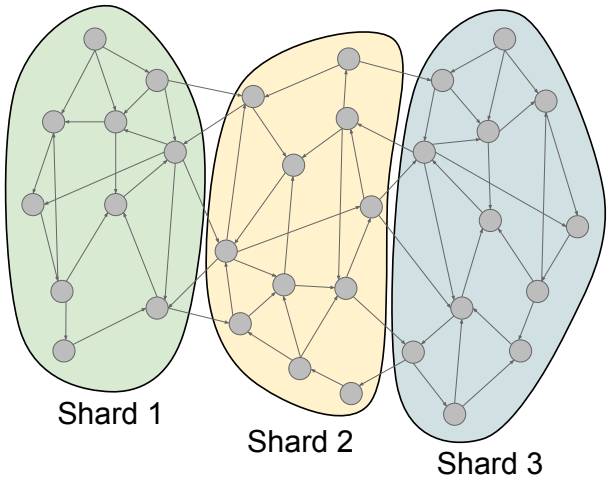

Figure 6. Unlike trees, which can be fairly trivially sharded with minimal cross-shard edges, the graphs used for graph-based ANN indices do not have this property. Sharding a single, globally trained, graph leads to many cross-shard edges and high communication overhead.

Standard graph partitioning algorithms are not only computationally intensive (often NP-hard for optimal solutions) but their heuristics may not be well-suited to the specific structural properties and traversal dynamics of ANN proximity graphs. For instance, the long-range links crucial for efficient greedy search in many graph-based ANN algorithms are particularly problematic; if these links frequently span across different shards, the very mechanism that enables fast convergence to the nearest neighbor is undermined by network delays.

As a result, the communication overhead from frequent cross-shard traversals can become a critical bottleneck. Each step in the graph walk that crosses a shard boundary requires fetching data (node features and neighbors) from a remote machine. If a sufficiently high number of edges are cross-shard edges, then more time will be spent on network overhead than on actual ANN computation. This problem increases in magnitude as the dataset, and therefore the number of shards, grows, because the proportion of cross-shard edges grows as the graph is partitioned more finely.

Furthering the network overhead problem is the relatively large amount of state that must be passed in every cross-shard request. Unlike the k-means trees used in partitioning-based algorithms, graphs are typically full of cycles and have no natural topological ordering. A cross-shard edge must therefore not only specify the receiving vertex and the query q, but also somehow encode information to prevent walking indefinitely in a cycle. In the single-machine case, this is handled by keeping track of visited vertices, but in a distributed setting, passing the visited set between machines can be prohibitively expensive as the visited set grows.

Consequently, while the notion of a distributed, globally indexed graph is theoretically attractive for preserving sublinear query cost scaling, the fine-grained exploration dependencies inherent in this style of algorithm likely preclude a practical implementation of such a style ANN index, and no competitive, globally indexed graph ANN algorithm has been created thus far.

8. Conclusion

ANN search efficiency is a multifaceted objective where different contexts lead to different optima. Approximate algorithms scale roughly with the cube root of dataset size, but require superlinear amounts of indexing cost to do so. Brute force linear scans have linear query cost *and* indexing cost, both with small constant factors, which can be advantageous in situations with small datasets or particularly high index refresh rates.

Our work highlights the importance of algorithmic advances that could reduce the need to trade off between query-time and indexing-time efficiency. Additionally, so long as this tradeoff exists, ANN libraries could improve their usability by better assisting users in choosing the most suitable algorithmic approach for their needs.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Andoni, A., Indyk, P., Laarhoven, T., Razenshteyn, I., and Schmidt, L. Practical and optimal lsh for angular distance. In *Proceedings of the 29th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, pp. 1225–1233, Cambridge, MA, USA, 2015. MIT Press.

Arya, S. and Mount, D. M. Approximate nearest neighbor queries in fixed dimensions. In *Proceedings of the*

Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '93, pp. 271–280, USA, 1993. Society for Industrial and Applied Mathematics. ISBN 0898713137.

Aumüller, M., Bernhardsson, E., and Faithfull, A. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms, 2018. URL https://arxiv.org/abs/1807.05614.

Chern, F., Hechtman, B., Davis, A., Guo, R., Majnemer, D., and Kumar, S. Tpu-knn: K nearest neighbor search at peak flop/s. In Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., and Oh, A. (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 15489–15501. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/639d992f819c2b40387d4d5170b8ffd7-Paper-Conference.pdf.

Dasgupta, S. and Freund, Y. Random projection trees and low dimensional manifolds. In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, STOC '08, pp. 537–546, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580470. doi: 10.1145/1374376.1374452. URL https://doi.org/10.1145/1374376.1374452.

Dasgupta, S. and Sinha, K. Randomized partition trees for exact nearest neighbor search. In Shalev-Shwartz, S. and Steinwart, I. (eds.), *Proceedings of the 26th Annual Conference on Learning Theory*, volume 30 of *Proceedings of Machine Learning Research*, pp. 317–337, Princeton, NJ, USA, 12–14 Jun 2013. PMLR. URL https://proceedings.mlr.press/v30/Dasgupta13.html.

Douze, M., Guzhva, A., Deng, C., Johnson, J., Szilvasy, G., Mazaré, P.-E., Lomeli, M., Hosseini, L., and Jégou, H. The faiss library, 2025. URL https://arxiv.org/abs/2401.08281.

Dwyer, R. A. Higher-dimensional voronoi diagrams in linear expected time. In *Proceedings of the Fifth Annual Symposium on Computational Geometry*, SCG '89, pp. 326–333, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 0897913183. doi: 10.1145/73833.73869. URL https://doi.org/10.1145/73833.73869.

Guo, R., Sun, P., Lindgren, E., Geng, Q., Simcha, D., Chern, F., and Kumar, S. Accelerating large-scale inference with anisotropic vector quantization. In III, H. D. and Singh, A. (eds.), *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 3887–3896. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/guo20h.html.

Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., de Las Casas, D., Hendricks, L. A., Welbl, J., Clark, A., Hennigan, T., Noland, E., Millican, K., van den Driessche, G., Damoc, B., Guy, A., Osindero, S., Simonyan, K., Elsen, E., Rae, J. W., Vinyals, O., and Sifre, L. Training compute-optimal large language models, 2022. URL https://arxiv.org/abs/2203.15556.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. Scaling laws for neural language models, 2020. URL https://arxiv.org/abs/2001.08361.

Malkov, Y. A. and Yashunin, D. A. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 42(4):824–836, April 2020. ISSN 0162-8828. doi: 10.1109/TPAMI.2018.2889473. URL https://doi.org/10.1109/TPAMI.2018.2889473.

Muennighoff, N., Yang, Z., Shi, W., Li, X. L., Fei-Fei, L., Hajishirzi, H., Zettlemoyer, L., Liang, P., Candès, E., and Hashimoto, T. s1: Simple test-time scaling, 2025. URL https://arxiv.org/abs/2501.19393.

Navarro, G. Searching in metric spaces by spatial approximation. In *6th International Symposium on String Processing and Information Retrieval. 5th International Workshop on Groupware (Cat. No.PR00268)*, pp. 141–148, 1999. doi: 10.1109/SPIRE.1999.796589.

Sharma, U. and Kaplan, J. A neural scaling law from the dimension of the data manifold, 2020. URL https://arxiv.org/abs/2004.10802.

Subramanya, S. J., Devvrit, Kadekodi, R., Krishaswamy, R., and Simhadri, H. V. *DiskANN: fast accurate billion-point nearest neighbor search on a single node*. Curran Associates Inc., Red Hook, NY, USA, 2019.

Sun, P., Simcha, D., Dopson, D., Guo, R., and Kumar, S. Soar: Improved indexing for approximate nearest neighbor search. In Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., and Levine, S. (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 3189–3204. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/0973524e02a712af33325d0688ae6f49-Paper-Conference.pdf.

A. Appendix

A.1. Query Compute Cost Regression: Experimental Setup

The Faiss performance numbers were all taken from Section 5.1 of (Douze et al., 2025). Douze et al. used various sized samples of BigANN (SIFT) vectors to obtain their results.

ScaNN and DiskANN numbers were taken from three benchmark sources:

1. Glove (1M) from `ann-benchmarks` (Aumüller et al., 2018).
2. Text2Image (10M) from the OOD track of the 2023 version of `big-ann-benchmarks`.
3. MS-Turing (1B) from the 2021 version of `big-ann-benchmarks`.