Unleashing Graph Partitioning for Large-Scale Nearest Neighbor Search

Lars Gottesbüren
Karlsruhe Institute of Technology
lars.gottesbueren@kit.edu

Laxman Dhulipala
University of Maryland, Google Research
laxman@umd.edu

Rajesh Jayaram
Google Research
rkjayaram@google.com

Jakub Łącki
Google Research
jlacki@google.com

ABSTRACT

We consider the fundamental problem of decomposing a large-scale approximate nearest neighbor search (ANNS) problem into smaller sub-problems. The goal is to partition the input points into neighborhood-preserving *shards*, so that the nearest neighbors of any point are contained in only a few shards. When a query arrives, a *routing* algorithm is used to identify the shards which should be searched for its nearest neighbors. This approach forms the backbone of distributed ANNS, where the dataset is so large that it must be split across multiple machines.

In this paper, we design simple and highly efficient routing methods based on clustering and locality-sensitive hashing. We prove strong theoretical guarantees for the LSH-based method, whereas the clustering-based method exhibits better empirical performance. A crucial characteristic of our routing algorithms is that they are inherently modular, and can be used with *any* partitioning method. This addresses a key drawback of prior approaches, where the routing algorithms are inextricably linked to their associated partitioning method. In particular, due to their modular structure, our routing methods enable the use of *balanced graph partitioning*, which is a high-quality partitioning method without a naturally associated routing algorithm. Prior routing methods compatible with graph partitioning are too slow to train on large-scale data.

We provide the first routing methods that are simultaneously compatible with graph partitioning, fast to train, admit low latency, and achieve high recall. In a comprehensive evaluation of our partitioning and routing on billion-scale datasets, we show that our methods outperform existing scalable partitioning methods by significant margins, achieving up to 1.72× higher QPS at 90% 10-recall than the best competitor and 1.27× in the geometric mean. Through fast and modular routing we establish graph partitioning as the new method of choice for partitioning large-scale ANNS datasets.

PVLDB Reference Format:
Lars Gottesbüren, Laxman Dhulipala, Rajesh Jayaram, and Jakub Łącki. Unleashing Graph Partitioning for Large-Scale Nearest Neighbor Search. PVLDB, 18(6): 1649 - 1662, 2025.
doi:10.14778/3725688.3725696

PVLDB Artifact Availability:
The source code is available at https://github.com/larsgottesbueren/gp-ann.

Proceedings of the VLDB Endowment, Vol. 18, No. 6 ISSN 2150-8097.
doi:10.14778/3725688.3725696

1 INTRODUCTION

Nearest neighbor search is a fundamental algorithmic primitive that is employed extensively across several fields including computer vision, information retrieval, and machine learning [42]. Given a set of points P in a metric space $(\mathcal{X}, d)$ (where $d\colon \mathcal{X} \times \mathcal{X} \mapsto \mathbb{R}_{\geq 0}$ is a distance function between elements of $\mathcal{X}$), the goal is to design a data structure which can quickly retrieve from P the k closest points from any query point $q \in \mathcal{X}$. The typical quality measure of interest is k*-recall*, which is defined as the fraction of true k nearest neighbors found.

Solving the problem with perfect recall in high-dimensional space effectively requires a linear scan over P in practice and in theory [2]. Therefore, most work has focused on finding *approximate* nearest neighbors (ANNS). The most successful methods are based on quantization [24, 31] and pruning the search space using an index data structure to avoid exhaustive search. Widely employed index data structures are k-d trees, k-means trees [8, 36, 38], graph-based indices such as DiskANN/Vamana [44] or HNSW [35] and flat inverted indices [13, 15, 28, 31, 34]. For a comprehensive review we refer to one of the several surveys on ANNS [1, 6, 33].

While graph indices offer the best recall versus throughput trade-off, they are restricted to run on a single machine due to fine-grained exploration dependencies and thus communication requirements. For a distributed system, the currently best approach is to decompose the problem into a collection of smaller ANNS problems that fit in memory [13]. The decomposition starts by performing *partitioning* – splitting the set P into some number s of subsets $\cup_{i=1}^{s} S_i = P$ (the *shards*). Then, to query for a point $q \in \mathcal{X}$, a lightweight *routing* algorithm is used to identify a small subset of the shards to search for the nearest neighbors of q. The search within a shard is then performed using an in-memory solution such as a graph index if the shards are large (e.g., in distributed ANNS), or exhaustive search if the shards are small. Optimizing the partition is crucial to achieving good query performance, because it allows querying only a small subset of the shards.

The partitioning and routing methods are closely connected, and, in some cases, inextricably intertwined. As a result, many existing routing methods require a particular partitioning method to be used. For example, consider partitioning via k-means clustering. Each shard (a k-means cluster) is naturally represented by the cluster center. The folklore *center-based routing* algorithm identifies the shards whose centers are closest to the query point to be searched, i.e., solves a much smaller ANNS problem first.

In a parameter regime where exhaustive search in a shard is feasible (e.g., $|S_i| \leq 10^5$) this approach is known as IVF (inverted file), another popular in-memory ANNS data structure [4, 24, 31]. For billion-scale datasets this entails $s \geq 10^4$ shards. In this paper, on the other hand, we are interested in investigating these methods for s in the order of $s \in [10, 100]$, with substantially more points per shard $|S_i| \approx 10^7 - 10^8$, as needed in the distributed setting.

We are interested in determining the best partitioning method for large-scale ANNS. In the following, we argue that despite being a very natural idea, the potential of *balanced graph partitioning* has remained untapped at billion-scale until now, due to several limitations.

The idea is to first compute a k-nearest neighbor graph $G = (V, E)$ of P and then compute a balanced partition of G with approximately minimal edge cut, which is a well-known NP-hard optimization problem with highly developed heuristic off-the-shelf solvers [7]. The k-nearest neighbor graph (k-NN) is a graph G whose vertices are points in P, in which each point has edges to its k nearest neighbors in P. The shards are computed by partitioning the vertices into roughly equal size sets, such that the number of edges which connect different sets is approximately minimized. More formally, compute a partition $\mathbf{\Pi}\colon V \to [s]$ of G into $s \in \mathbb{N}$ shards of size $|\mathbf{\Pi}^{-1}(i)| \leq \frac{(1+\varepsilon)|P|}{s}$ $\forall i \in [s]$, while minimizing $|\{(u, v) \in E \mid \mathbf{\Pi}(u) \neq \mathbf{\Pi}(v)\}|$. This attempts to put the maximum possible number of nearest neighbors of each vertex v in the same shard as v.

Recently, Dong et.al. [15] achieved good results with this method on million-scale data as an IVF data structure, beating the widely used k-means partitioning. To tackle the routing problem, they train a neural model after partitioning. Unfortunately, training their routing model is extremely slow even at million-scale, ruling out use at billion-scale. While partitioning with k-NN graphs promises significantly improved recall, it comes with two main limitations, which have hindered its adoption in favor of k-means partitioning.

(1) The shards do not admit natural geometric properties, such as convexity of shards computed by k-means clustering. As a result, there is no obvious method to route query points to shards. In fact, as mentioned, the best known routing method requires training computationally expensive neural models [15, 17].
(2) A k-NN graph is required to partition the pointset. This is computationally expensive and seemingly introduces a chicken and egg problem, as computing a k-NN graph requires solving the ANNS problem.

In this paper, we show how to address the above limitations and *unleash* the benefits of balanced graph partitioning for billion-scale ANNS problems.

Contribution 1: Fast, Inexact Graph Building Suffices for High-Quality Partitions. We empirically demonstrate that a very rough approximate k-NN graph built by recursively performing *dense ball carving* suffices to obtain high quality partitions and query performance, and that such a coarse approximation can be computed quickly.

Contribution 2: Fast, Accurate, and Modular Combinatorial Routing. We devise two combinatorial routing methods which are fast, high quality, and can be used with *any* partitioning method, in particular with graph partitioning, as they are trained on a finalized partition. Both adapt center-based routing to large shards, which is needed for distributed ANNS. Our first and empirically stronger routing method is called KRT. It is based on sub-clustering the points within each shard using hierarchical k-means. We use either the tree representation of the clustering or HNSW to retrieve center points, routing the query to the shards associated with the closest retrieved points. Our second method called HRT is based on locality sensitive hashing (LSH). It works by locating the query point in the sorted ordering of LSH compound hashes, and inspecting nearby points to determine which shards to query.

Contribution 3: Theoretical Guarantees for Routing. We provide a theoretical analysis of HRT, and establish rigorous guarantees on its performance. Specifically, we prove that each query point will be routed to a shard containing one of its approximate nearest neighbors with high probability. To the best of our knowledge, these are the first theoretical guarantees for the routing step.

While KRT performs better in practice, it does not offer theoretical guarantees, which is the benefit of HRT. This gap between theory and practice is common in approximate nearest neighbor search, where LSH-based methods provide better theoretical understanding of the problem, whereas clustering or graph-based methods perform better empirically due to being data-dependent.

Contribution 4: Empirical Evaluation. In our extensive evaluation, we demonstrate that shards obtained via balanced graph partitioning attain up to 1.72x higher throughput than the best competing partitioning method, and 1.27x in the geometric mean at billion-scale. While the partitioning time is 3× slower, the absolute time is reasonable: 2 hours for the highest-dimensional instance, giving a highly favorable trade-off. We analyze the shards and find that the concentration of ground-truth neighbors in the top-ranked shard per query is significantly higher (up to +25%).

Our routing methods are multiple orders of magnitude faster to train than the existing neural network based approaches [15, 17] while obtaining similar or better recall at similar or lower routing time. Specifically, our KRT method can be trained in half an hour on billion-point datasets, compared to multiple hours required by the neural network approaches on 1000x smaller million-point datasets where KRT training takes under a second.

2 PARTITIONING

We present two improvements to k-NN graph partitioning. To speed up the k-NN graph construction we present a highly scalable approximate algorithm in Section 2.1. Moreover, we propose an algorithm to compute overlapping shards in Section 2.2, to prevent recall losses in boundary regions. To compute the initial disjoint partition, we use the KaMinPar graph partitioner [23], for which we give a brief description in Section 5.

2.1 Approximate k-NN Graph-Building

To speed up k-NN graph building, we use a simple approximate algorithm based on recursive splitting with ***dense ball clusters***, as outlined in Algorithm 1. As long as the number of points is not sufficiently small for all-pairs comparisons, we split them using the following heuristic. We sample a small number of pivots L from the pointset, and assign each point to the cluster P_c represented by its closest pivot c. Ties are broken arbitrarily. The clusters are treated recursively, either splitting them again or generating edges for all pairs in the cluster, filtered down to the top k neighbors for each point. We take the union of the edges generated in different clusters.

Algorithm 1: Recursive Dense Ball Graph Construction

```
Input: Pointset $P$, all-pairs threshold $\alpha$, pivot sampling
       fraction $\beta$, maximum number of pivots $\gamma$
if $|P| \leq \alpha$
    Compute $d(P \times P)$ in parallel
    Emit top $k$ neighbors for each $p \in P$
else
    Sample $L \subset P$ uniformly at random with
     $|L| = \min(\beta \cdot |P|, \gamma)$
    Compute $d(P \times L)$ in parallel
    for $c \in L$ do in parallel
        $P_c \longleftarrow \{p \in P \mid d(p, c) = \min_{l \in L} d(p, l)\}$
        Recurse on $P_c$
```

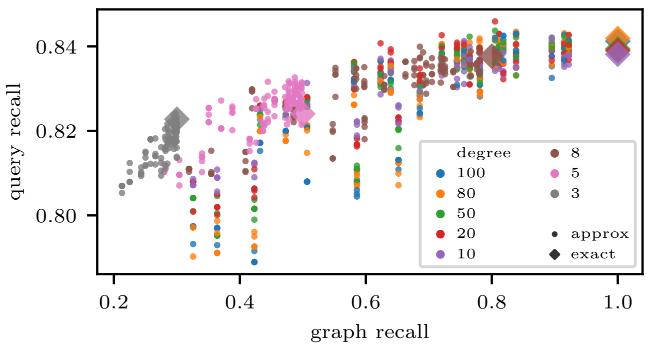

Figure 1: The x-axis shows the recall of the approximate k-NN graph used for partitioning. The y-axis shows the average recall of queries for 10 nearest neighbors when only a single shard is inspected. The plot uses a 1M slice of the DEEP dataset.

The intuition is that top-k neighbors will be clustered together with good probability.

We use a constant threshold α below which we deem a cluster sufficiently small for all-pairs comparison. The number of pivots $l = |L|$ is determined with a parameter $\beta \in (0, 1)$ and a large constant γ (around 1000), as $l = \min(\beta \cdot |P|, \gamma)$.

Improved graph quality is attained via r independent repetitions, and assigning points to the f closest pivots instead of just one. The latter can only be done a few times without increasing running time exponentially, as this replicates each point in the current cluster to f subclusters for recursion. We use $f > 1$ on the first recursion level and $f = 1$ at lower levels. We again take the union of generated edges from different repetitions and filter down to the top k for each point.

Using an approximate k-NN graph for partitioning the dataset is acceptable since only coarse local structure must be captured to ensure partition quality, i.e., the edges need only connect near neighbors together, not necessarily the exact top-k. Figure 1 demonstrates that even low-quality graphs (k-NN graph recall ≈ 0.3) lead to high query recall: more than 81% of top-10 neighbors are concentrated in one shard per query. Using an exact k-NN graph (100% recall) for partitioning only increases the query ground truth neighbor concentration to 84%. Here we use a slice of the DEEP dataset [3, 43] with $n = 10^6$ points partitioned into $s = 16$ shards of size $|S_i| \leq 65625$ each ($\varepsilon = 5\%$). The x-axis shows the recall of the k-NN graph; the y-axis shows the average fraction of top-10 ground-truth neighbors of a query in the shard with most ground-truth neighbors (the concentration). To obtain approximate graphs with different quality scores, we vary the number of repetitions $r \in \{2, 3, 5, 8, 10\}$, top-level fanout $f \in \{2, 3, 5, 8, 10\}$, cluster size threshold $\alpha \in \{500, 1000, 2000, 5000, 10000\}$ and the degree $\in \{3, 5, 8, 10, 20, 50, 80, 100\}$. The pivot sample fraction β is a constant set to $\beta = 0.005$.

We also observe that higher degrees may in some cases lead to slightly worse query recall. We suspect that since we use a low-quality method, adding more approximate neighbors pollutes the graph structure. Note that this effect does not appear with an exact k-NN graph computed via all-pairs comparison (marked as diamond in the plot). Notably, the query performance between approximate and exact graphs differs only by a small margin. Overall, these observations justify the use of highly sparse approximate graphs with only a few edges per point for the partitioning step, without sacrificing much of the query performance.

2.2 Partitioning into Overlapping Shards

When partitioning into disjoint shards, we can incur losses on points on the boundary between shards, whose k-NNs straddle multiple shards. To address this issue, we propose a greedy algorithm inspired by local search for graph partitioning [18] which eliminates cut edges by replicating nodes in a post-processing step. The set of cut (k-NN) edges is precisely the recall loss when the points themselves are queries [15], thus minimizing cut edges is a natural strategy in our setting.

We introduce a new parameter $o \geq 1$ to restrict the amount of replication. For a fair comparison with disjoint shards in memory-constrained settings, we keep the maximum shard size $L_{\max}(s)$ fixed and instead increase the number of shards to $s' = o \cdot s$. We first compute a disjoint partition into s' shards of size $\frac{(1+\varepsilon)|P|}{s'}$ and then apply an overlap algorithm with $\frac{(1+\varepsilon)|P|}{s}$ as the final size.

In each step, our ***overlap algorithm*** takes a node u and places it in the shard S_i that contains the plurality of its neighbors and not u. This increases the average recall by $\frac{\text{cut}(u,S_i)}{k|P|}$, where $\text{cut}(u, S_i) = |\{(u, v) \in E \mid v \in S_i\}|$ for all S_i with $u \notin S_i$. We repeat this procedure until there is no more placement into a below size-constraint shard which removes at least one edge from the cut. We greedily select the node whose placement eliminates the most cut edges for the next step.

To parallelize this seemingly sequential procedure, we observe that nodes whose placement removes the same number of cut edges can be placed at the same time, i.e., grouped into bulk-synchronous rounds. Moreover, the number of possible cut values is small; at most k. Thus, only few rounds are necessary since each node removes at least one cut edge and no new cut edges are added.

3 ROUTING

The goal of routing is to identify a *small number of* shards which contain a large fraction of the k nearest neighbors of a given query $q \in \mathcal{X}$. We adapt center-based routing for our purpose. When using

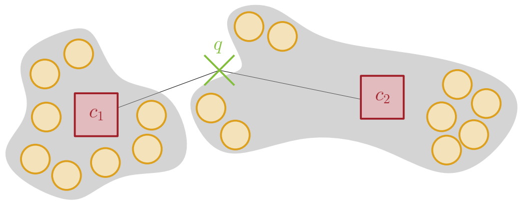

Figure 2: Illustration of an example where routing using a single center per shard fails. The nearest neighbors of q are in the cluster of c_2, but $d(q, c_1) < d(q, c_2)$. If the hierarchical sub-clusters are represented with their own centers, the routing works correctly.

IVF for $|P| = 10^9$ points, a typical configuration uses $s \approx 10^6$ shards, with $|S_i| \approx 10^3$ points each [4]. Using a single center per shard works well in this setup. In contrast, in our setup we have few shards $s \in [10, 100]$ of large size $|S_i| > 10^7$ and use a graph index per shard to retrieve neighbors.

Large shards cannot be as easily represented with a single center. Figure 2 illustrates the problem where routing fails because sub-cluster structures are not represented well by the center. Moreover, the shards from graph partitioning are not connected convex regions as in k-means clustering, which further aggravates this issue. For example, routing with a single center resulted in only 28% 10-recall in the top-ranked shard on the MS Turing dataset, as opposed to 81% with our best routing index.

Instead of a single center, we represent each shard $S_i \subset P$ by multiple points R_i with the goal of accurately capturing sub-cluster structure. At query time, given a query point $q \in \mathcal{X}$ we retrieve from $R = \cup_{i \in [s]} R_i$ a set Q of (approximately) closest points from q. Then, we route the query to the shards which the points of Q belong to. More precisely, we compute a probe order of the shards, ranking each shard i based on the distance between q and the closest point in $Q \cap R_i$. The size of R is limited to a parameter m so that the routing index fits in memory.

Our routing algorithms differ in the way the coarse representatives R_i are constructed at training time and by the ANNS retrieval data structure used to determine Q. The first algorithm uses hierarchical k-means to coarsen P to R and uses the resulting k-means tree for retrieval. This method is called ***KRT*** for k-means RoutTing. In the experiments we also build an HNSW index on all points of R for even faster retrieval. The second algorithm called ***HRT*** (HashRouTing) uses uniform random sampling to coarsen and a variant of LSH Forest [5] to find Q. In Section 3.3 we provide theoretical bounds for the quality of HRT, and later demonstrate that both methods perform well in practice, with KRT having a sizeable advantage empirically.

3.1 K-Means Tree Routing Index: KRT

Algorithm 2 shows the training stage for KRT. We create one root node v_i per shard S_i and hierarchically construct a separate k-means tree for each shard. The set R_i consists of centroids across all recursion levels of hierachical k-means. Per level we use $l = 64$ centroids for the sub-tree roots. Furthermore, we are given a maximum index size m, which we split proportionately among sub-trees according to their cluster size. We subtract l from the budget on each level, to account for the centroids. The recursive coarsening stops once the budget m is exceeded or the number of points is below a cluster size threshold λ (we use 350). In contrast to usual k-means search trees, we do not store or search the points in leaf-nodes, as our goal is to coarsen the dataset.

Algorithm 2: KRT: Training

```
Input: number of shards $s$, shards $S_i$, number of centroids $l$,
       index size $m$, cluster size $\lambda$
for $i = 1$ to $s$ do in parallel
    create root node $v_i$
    Build($S_i, l, \frac{|S_i|(m-s)}{|P|}, \lambda, v_i$)
func Build($P, l, m, \lambda, v$):
    if $m \le 1$ return
    centroids($v$) ← K-Means($P, l$)
    for $c \in$ centroids($v$) do in parallel
        $P_c$ ← $k$-means cluster of $c$
        create new tree-node $v_c$
        add tree-edge from parent $v$ to $v_c$
        if $|P_c| > \lambda$
            Build($P_c, l, \frac{(m-l)|P_c|}{|P|}, \lambda, v_c$)
```

Algorithm 3: KRT: Routing

```
Input: Search budget $b$, query point $q$
PQ ← $\{(v_i, 0, i) \mid i \in [s]\}$ // min-heap with (tree node, key,
  shard ID) prioritized by key
min-dist[$i$] ← $\infty$   $\forall i \in [s]$
while PQ not empty and $b-- > 0$ do
    $(v, d_v, s_v)$ ← PQ.pop()
    for $c \in$ centroids($v$) do
        min-dist[$s_v$] ← min(min-dist[$s_v$], $d(q, c)$)
        if $c$ has a sub-tree
            add $(v_c, d(q, c), s_v)$ to PQ
return sort $[s]$ by min-dist
```

Algorithm 3 shows the routing algorithm which is similar to beam-search [41]. At a non-leaf node we score the centroids against the query q and insert the non-leaf children with the associated centroid distance into a priority queue for further exploration. Initially the priority queue contains the tree roots. The search terminates when either the priority queue is empty or a search budget b (a parameter) is exceeded.

3.2 Frequency-Based Routing

In addition to routing by minimum distance, we propose a second signal that is good at routing: the frequency of how often a shard appears in Q. Let w be a second beam-width parameter and let $Q' \subset Q$ be the w closest points to the query from Q. Then the priority of shard S_i is $|\{v \in Q' \mid v \text{ is in subtree rooted at } v_i\}|$, and shards are sorted by decreasing priority. For KRT with HNSW acceleration, we use $Q' = Q$, i.e., use the same beam-width w = ef_search as HNSW.

Algorithm 4: hRt: Training

```
Input: Shards $S_i$, index size $m \leq n$, repetition parameter $r$,
       sketch size $t$
for $i = 1$ to $s$ do in parallel
    Sample $\lfloor \frac{m \cdot |S_i|}{|P|} \rfloor$ points $R_i$ from $S_i$ uniformly without
      replacement
$R = \cup_{i=1}^{s} R_i$
for $j = 1$ to $r$ do in parallel
    Draw independent random hash functions $h_{j,1}, \ldots, h_{j,t}$
      from a LSH family for $\mathcal{X}$
    For each $x \in R$ define a sorting key
      $h_j(x) = (h_{j,1}(x), \ldots, h_{j,t}(x))$
    Sort $R$ lexicographically with keys $h_j$ to obtain $I_j$
```

Algorithm 5: hRt: Routing

```
Input: Query point $q$, window size $W \geq 1$, partition
       $\Pi' : R \rightarrow [s]$
min-dist$[i] \leftarrow \infty \quad \forall i \in [s]$
for $j = 1$ to $r$ do
    Binary search in $I_j$ for position $\tau$ of the point
      lexicographically closest to $h_j(q)$
    for $c \in I_j[\tau - W : \tau + W]$ do
        min-dist$[\Pi'(c)] \leftarrow \min($min-dist$[\Pi'(c)], d(q, c))$
return sort $[s]$ by min-dist
```

However, for plain kRt, the set Q contains too many irrelevant points, as it is yet not filtered by distance during the search.

Empirically, we found that frequency-based routing is good at finding the first-ranked shard to probe, but performs worse than distance-based routing at ranking later probed shards. Therefore, we also consider a hybrid approach, where the first shard to probe is determined using frequency-based routing and the remaining shards are ranked by distance.

3.3 Sorting-LSH Routing Index: hRt

We now describe a routing scheme based on *Locality Sensitive Hashing* (LSH). At a high level, an LSH family $\mathcal{H}$ is a family of hash functions of the form $h : \mathcal{X} \rightarrow \{0, 1\}$, such that similar points are more likely to collide; namely, the probability $\mathbf{Pr}_{h \sim \mathcal{H}}[h(x) = h(y)]$ should be large when $x, y \in \mathcal{X}$ are similar, and small when x, y are farther apart. We formalize this in the proof of Theorem 1, and describe the routing index assuming we have an LSH family.

Algorithm 4 describes the construction of a SortingLSH index. We first subsample m points R from P uniformly at random. A single SortingLSH index is created as follows: we hash each point $x \in R$ multiple times via independent hash functions from $\mathcal{H}$, and concatenate the hashes into a string $(h_1(x), h_2(x), \ldots, h_t(x))$. Importantly, we use the same hash functions $h_1, \ldots, h_t$ for each point in R. We *sort* the points in R *lexicographically based on these strings of hashes.* The index is the points stored in sorted order along with their hashes. Intuitively, similar points are more likely to collide often, thus share a longer prefix in their compound hash string, and thus are more likely to be closer together in the sorted order. We repeat the process r times to improve retrieval quality (up to 24 in our experiments).

Next, Algorithm 5 describes the routing procedure for SortingLSH indices. Given a query point q, for each of the r indices, we hash q with the LSH functions used for that repetition, compute the compound hash string $h(q) = (h_1(q), \ldots, h_t(q))$ for q, and then find the position $\tau \in [m]$ that is lexicographically closest to $h(q)$. We retrieve the window $[\tau - W, \tau + W]$ in the sorted order to consider these points' distances to the query in the ranking.

One key advantage of employing LSH is that we can prove formal guarantees for the retrieved points. Since our approach is more involved than searching through a single set of hash buckets, we provide a new analysis which demonstrates that it provably recovers a set of relevant points Q that results in routing to a shard which contains an approximate nearest neighbor. We remark that, while the routing only guarantees we are routed to a shard containing an *approximate* nearest neighbor, under mild assumptions on the partition, most of the approximate nearest neighbors of a given point will be in the same shard as the closest point. Under such a natural assumption, our SortingLSH index also recovers the true nearest neighbors.

Theorem 1. *Fix any approximation factor $c > 1$, and let $P \subset (\mathbb{R}^d, \|\cdot\|_\rho)$ be a subset of the d-dimensional space equipped with the ℓ_ρ norm, for any $\rho \in [1, 2]$. Let $n = |P|$ and m denote the routing index size. Set stretch factor $\alpha = O(c)$, repetitions $r = O(n^{1/c})$ and window size $W = O(1)$. Denote by $\pi_k(q)$ the distance from q to the k-th nearest neighbor of q in P. Then the following holds for any query $q \in \mathbb{R}^d$:*

- *If $m = n$, then with probability $1 - 1/\mathrm{poly}(n)$ q gets routed to a shard $\mathbf{\Pi}(p)$ containing at least one point $p \in P$ that satisfies $\|p - q\|_\rho \leq \alpha\pi_1(q)$.*
- *If $m < n$, then for any $\delta \in (0, 1)$, with probability $1 - \delta$ the query q gets routed to a shard $\mathbf{\Pi}(p)$ containing at least one point $p \in P$ that satisfies $\|p - q\|_\rho \leq \alpha \cdot \pi_{\lceil \log \delta^{-1} \frac{n}{m} \rceil}(q)$.*

Proof. The proof consists of the following steps: 1) embedding into Hamming space, 2) constructing a hash family and SortingLSH index for Hamming space, 3) analyze the prefix collision probabilities and lastly 4) apply these to routing decisions.

To simplify the construction of the hash family $\mathcal{H}$, we first embed the pointset P into a subset of the d'-dimensional Hamming cube $\{0, 1\}^{d'}$ with a constant distortion in distances. Let $\Phi = \frac{\max_{x,y \in P} \|x-y\|_\rho}{\min_{x,y \in P} \|x-y\|_\rho}$ denote the aspect ratio of P. By Lemma A.2 and A.3 of [10], for any $\rho \in [1, 2]$ there exists an embedding $f_\rho : P \rightarrow \{0, 1\}^{d'}$, with $d' = O(d\Phi \log n)$, such that there exists a constant C so that with probability $1 - 1/\mathrm{poly}(n)$ for all $x, y \in P$, we have $\|f_\rho(x) - f_\rho(y)\|_0 \leq \|x - y\|_\rho \leq C \cdot \|f_\rho(x) - f_\rho(y)\|_0$, where $\|a - b\|_0 = |\{i \in [d'] | a_i \neq b_i\}|$ is the Hamming distance between any two $a, b \in \{0, 1\}^{d'}$. The constant C will go into the approximation factor of the retrieval, but note that C can be set to $(1 + \varepsilon)$ by increasing the dimension by a $O(1/\varepsilon^2)$ factor. Thus, in what follows, we may assume that $P \subset \{0, 1\}^{d'}$ as well as the query q are a subset of the d'-dimensional hypercube equipped with the Hamming distance.

We now construct the hash family $\mathcal{H}$ which we will use for the SortingLSH Index. A draw $h \sim \mathcal{H}$ is generated as follows: (1) First, sample $i_1, \dots, i_\ell \sim [d']$ uniformly at random, where $\ell = O(d' \log n)$, (2) for any $x \in \{0,1\}^{d'}$ define $h(x) = (x_{i_1}, x_{i_2}, \dots, x_{i_\ell}) \in \{0,1\}^\ell$. As notation, for any $\ell' \le \ell$ and hash function h defined in the above way, we write $h_{\ell'}(x) = (x_{i_1}, x_{i_2}, \dots, x_{i_{\ell'}}) \in \{0,1\}^{\ell'}$ to denote the ℓ'-prefix of h.

We are now ready to prove the main claims of the Theorem. First, suppose we are in the case that $m = n$, and thus the pointset is *not* subsampled before the construction of the SortingLSH index. Let $p^* = \arg\min_{p \in R} \|p - q\|_0$ be the nearest neighbor to q, with ties broken arbitrarily. We first claim that p^* and q share a t-length prefix in at least one of the $i \in [r]$ repetitions of SortingLSH. Set $t^* = \frac{4d' \ln n}{c\|p^* - q\|_0}$. Then the probability that p^*, q share a prefix of length at least t^* – namely, the event $h_{t^*}(p^*) = h_{t^*}(q)$, is at least

$$\left(1 - \frac{\|p^* - q\|_0}{d'}\right)^{t^*} = \left(1 - \frac{\|p^* - q\|_0}{d'}\right)^{\frac{(4/c)d' \log n}{\|p^* - q\|_0}} \ge \left(\frac{1}{2}\right)^{(8/c)\log n}$$
$$= n^{O(1/c)}$$

where we used the inequality $(1 - x/2)^{1/x} \ge 1/2$ for any $0 < x < 1$. Next, for any x such that $\|x - q\|_0 \ge 10/c\|p^* - q\|_0$, we have:

$$\left(1 - \frac{\|x - q\|_0}{d'}\right)^{t*} = \left(1 - \frac{\|x - q\|_0}{d'}\right)^{\frac{(1/c)d' \log n}{\|p^* - q\|_0}} \le 1/n^4$$

where we used the inequality that $(1 - x)^{n/x} \le (1/2)^n$ for any $x \in (0,1]$ and $n \ge 1$. Union bounding over all $r < n$ trials, it follows that with probability at least $1 - 1/n^3$, we never have $h_{t^*}(x) = h_{t^*}(q)$ for any x such that $\|x - q\|_0 \ge 10/c\|p^* - q\|_0$, and moreover we do have $h_{t^*}(p^*) = h_{t^*}(q)$ for at least one repetition of the sampling.

Let $h^1, h^2, \dots, h^r$ be the r independent hash functions drawn for the SortingLSH routing index. By the above, there exists a repetition $i \in [r]$ such that $h^i_{t^*}(p^*) = h^i_{t^*}(q)$.

First, suppose that p^* was added to the window. Then since p^* is the closest point to q, by construction of the algorithm the point q will be deterministically routed to $\mathbf{\Pi}[p']$ for a point p' such that $\|q - p'\|_\rho = \|q - p^*\|_\rho$, which completes the proof in this case. Otherwise, on repetition i, if p^* was not added to the window, then since $h^i_{t^*}(p^*) = h^i_{t^*}(q)$, there must be another point p' with $h^i_{t^*}(p') = h^i_{t^*}(q)$ on that repetition. By the above, such a point must satisfy $\|q - p'\|_\rho \le O(c)\|q - p^*\|_\rho$ as needed.

Finally, the case of $m < n$ follows by noting that the $O(\log(1/\delta) \cdot n/m)$-th nearest neighbor to q will survive after sub-sampling m points from n with probability at least $1 - \delta/2$, in which case the above analysis applies.

□

4 EMPIRICAL EVALUATION

We evaluate the performance of partitioning-based approximate nearest neighbor search algorithms using different partitioning and routing algorithms in terms of the recall of retrieving top-10 nearest neighbors versus number of shards probed η and throughput (queries-per-second). In order to assess the quality of partitions alone, we additionally look at recall with a *hypothetical* routing oracle that knows the entire dataset and picks an optimal sequence of shards to probe, i.e., one that maximizes recall.

4.1 Experimental Setup

Datasets. We test on the kNN datasets from https://big-ann-benchmarks.com/neurips21.html, which have one billion points each; namely *DEEP* (dim = 96, ℓ_2), *Text-to-Image* (dim = 200, inner product, abbreviated as TTI) and *Turing* (dim = 100, ℓ_2) which have real-valued coordinates, as well as SIFT1B (dim = 128, ℓ_2, also known as BIGANN) and SpaceV (dim = 100, ℓ_2) which have byte-valued coordinates. These are some of the *most challenging billion-scale ANNS datasets* that are currently being evaluated by the community. For example Text-to-Image exhibits out-of-distribution query characteristics, as demonstrated in [30].

We allow $\varepsilon = 5\%$ imbalance of shard sizes. In addition to disjoint partitions, we also consider overlapping partitions with 20% replication ($o = 1.2$). We use $s = 40$ shards in the non-overlapping case and $s = 48$ shards in the overlapping case (so that the maximum shard size is the same in both cases).

As the neural routing methods do not scale to billions of points, we also benchmark on two smaller datasets, namely SIFT1M (dim = 128, ℓ_2) and GLOVE (dim = 100, angular distance), which have roughly a million points, partitioned into $s = 16$ shards with $\varepsilon = 5\%$.

Machine Setup. The query experiments are run on a 64-core (1 socket) EPYC 7702P clocked at 2GHz with 1TB RAM and 256 MB L3 cache. The partitioning experiments are run on a 128-core (2 sockets, 16 NUMA nodes) EPYC 7713 clocked at 1.5GHz with 2TB RAM and 256 MB L3 cache. The small-scale experiments comparing GP with the neural methods are run on a 128-core EPYC 7742 clocked at 2.25GHz with 1TB RAM and 256 MB L3 cache.

4.2 Methods

We compare graph partitioning (GP) with three prior scalable partitioning methods: k-means (KM) clustering, balanced k-means (BKM) clustering [11], and Pyramid [13]. To distinguish disjoint and overlapping partitioning we prefix the corresponding method name with O (OGP, OKM, OBKM). OKM and OBKM create overlap by assigning points to second-closest clusters (etc.) as proposed by [8]. We implemented our algorithms and baselines in a common framework in C++ for a fair comparison. Our source code is available at https://github.com/larsgottesbueren/gp-ann. We note that the original source code for Pyramid [13] is not available, and the original source code for BKM [11] is not parallel, which is prohibitive for us. Thus we implemented a parallel version of BKM.

To compute graph partitions, we use KaMinPar [23] which is the currently fastest algorithm. For an overview of the field of graph partitioning, we refer to a recent survey [7] and for a brief description of KaMinPar, we refer to Section 5. The k for the k-NN graph used in graph partitioning is set to 10.

Neural Routing. Later on, we also compare with two methods that use neural routing: BLISS [25] and USP [17]. We exclude Neural-LSH [15] since it is strictly outperformed by USP in all aspects, but remark that its training took over two days on GLOVE. USP and BLISS jointly learn the partition and routing index using neural

Table 1: Imbalance and 10-recall for Pyramid at $\eta = 1$.

dataset	algorithm	max shard size	routing oracle recall $\eta = 1$
DEEP	Pyramid imbalanced	34.4M	80.3%
DEEP	Pyramid reassign	26.25M	77.2%
DEEP	Pyramid bin-packing	26.2M	68.8%
Turing	Pyramid imbalanced	31.2M	75.9%
Turing	Pyramid reassign	26.25M	75.2%
Turing	Pyramid bin-packing	26.21M	70.6%
Text-to-Image	Pyramid imbalanced	47.2M	80.5%
Text-to-Image	Pyramid reassign	26.25M	76.9%
Text-to-Image	Pyramid bin-packing	26.01M	70.6%
SpaceV	Pyramid imbalanced	30.8M	87.4%
SpaceV	Pyramid reassign	26.25M	87.2%
SpaceV	Pyramid bin-packing	26.25M	82.1%
SIFT1B	Pyramid imbalanced	41.4M	70.8%
SIFT1B	Pyramid reassign	26.25M	68.1%
SIFT1B	Pyramid bin-packing	26.23M	59.6%

networks. Given a query, the neural net infers a probability distribution of the shards, which is interpreted as a probe order. BLISS uses a cross-entropy loss formulation to maximize the number of points which have at least one near-neighbor in their shard, and uses the power of k choices (top ranked shards) to achieve a balanced assignment. USP uses a linear combination of edge cut and normalized squared shard size as the loss function. Both methods rely on building a k-NN graph for their loss function. We use the publicly available Python implementations in PyTorch (USP) and TensorFlow (BLISS).

k-means Balancing. While k-means clusters are fairly balanced, they often do not adhere to the strict shard size limit. For KM, we tested two rebalancing approaches: 1) migrate points from overloaded clusters to their second closest center (etc.) and 2) splitting overloaded clusters recursively with k-means using smaller k. Remigrating is better for QPS which is why we opted for this approach.

Pyramid. In the following we describe the partitioning and routing used in Pyramid [12, 13]. First the dataset P is randomly subsampled to a smaller pointset P'. P' is then further aggregated via flat k-means to $\hat{P}$, with $|\hat{P}| = 10000$. On $\hat{P}$ a (HNSW) graph is built, which is partitioned into balanced shards using graph partitioning. This HNSW graph is used as the routing index – all shards with points visited in a query are probed. Hence, the search-beamwidth affects the number of probes. The partition of $\hat{P}$ is extended to P by assigning each point to the shard of its nearest neighbor in $\hat{P}$.

Because of the last assignment step, the shards are highly imbalanced. In Table 1 we report that Pyramid exhibits between 23.2% – 88.8% imbalance. If we enforce a 5% imbalance for Pyramid by reassigning points to the closest cluster below the size constraint, recall drops by 0.2% – 3.6%. We use this balanced version in the experiments. Another idea is to overpartition into more shards and perform bin-packing into balanced shards. We overpartition by 10x. As shown this method performs worse than reassignment.

Note that we achieved better partitions and query throughput by partitioning a k-NN graph on $\hat{P}$ rather than partitioning the HNSW graph. For routing we still use the HNSW graph built on $\hat{P}$.

Table 2: Parameter overview and default values for $|P| = 10^9$.

Parameter	Value	Description
α	5000	threshold for all-pairs
β	0.005	# pivots = $\min(\gamma, \beta n)$
γ	1500 | 950	max pivots. recursion | top level
r	3	graph-building repetitions
f	3	assign to f pivots on top-level
m	20K - 10M	routing index size
l	64	KRT centroids
λ	350	KRT recursion cutoff
W	50-4000	HRT window size
r	1 - 24	HRT repetitions
b	5000-50000	max comparisons for HRT and KRT

4.3 Configuration

In Table 2 we recap the different parameters of the graph-building and routing algorithms and provide sensible default or grid search values that were tuned for index building speed and good query recall at $|P| = 10^9$ points. The values for graph-building and routing index construction also work well for other pointset sizes. The routing query parameters (comparison budget, window size or beamwidth) should be tuned for each dataset. The routing index size m should be tuned offline with consideration for memory consumption and routing recall.

Partitioning Configuration. For k-NN graph building with dense ball clusters we use $r = 3$, $f = 3$, $\alpha = 5000$, $\beta = 0.005$. Larger values of α, r, f lead to higher quality graphs at the expense of higher running time, with linear dependence on r and f and quadratic dependence on α. As we showed earlier, we can obtain high query recall even with a low-recall graph. Therefore, these values are chosen to save on index building time. The number of pivots $l = |L|$ is set to $l = \min(1500, \beta|P|)$ on the lower recursion levels; and reduced to $l = \min(950, \beta|P|)$ on the top level to save running time. This value should balance between splitting the dataset into small clusters (large value) and retaining the sampled pivots in cache. We use the *default* preset of KaMinPar. For SIFT1M and GLOVE we use $r = 5$ and $f = 5$ as well as the *strong* preset of KaMinPar since the budget for index construction time is not as constrained as for billion-scale.

Routing Configuration for Billion-Scale. Unless mentioned otherwise, all methods use KRT for routing, since it is also the generalization of k-means' native routing method for large shards, which is the regime we are interested in. For the throughput evaluation we accelerate the routing query with HNSW. For k-means based partitioning methods, we also use the native routing method with one center per shard.

We use a cluster size threshold of $\lambda = 350$, number of centroids $l = 64$ and tree search budget $b = 50K$. In preliminary experiments, we tested different parameter settings and found the results were not sensitive to λ and l which is why these results are not reported here, whereas b does influence routing quality. Note that we do not explore different budgets b here, as KRT with tree-search is only used in Section 4.7. To explore different routing quality configurations, we therefore vary the size m of the set of coarse representatives R, testing $m \in \{20K, 100K, 200K, 500K, 1M, 2M, 5M, 10M\}$. Larger values of m lead to more accurate routing, at the expense of a linear increase in memory and sublinear increase in routing time. We observe that larger m also lead to higher QPS for all values we tested, though with diminishing returns.

The HNSW configuration for routing uses a degree of 32 and beamwidth ef_construction = 200 for insertion. To explore different routing quality trade-offs we vary the beamwidth during search ef_search $\in \{20, 40, 80, 120, 200, 250, 300, 400, 500\}$.

The memory consumption for the routing index is $m \cdot (dim + \deg + 1) \cdot 4$ bytes. For example, using $m = 500K, dim = 100$ and degree 32 leads to 266MB, or $m = 5M$ leads to 2.66GB.

Based on preliminary testing, we use distance-based routing for the datasets Turing, DEEP, and SpaceV, and we use the distance-frequency-hybrid for Text-to-Image and SIFT1B. Changing this parameter does not require rebuilding any data structures, hence it can be easily tuned at query time.

For routing with HRT we test the number of repetitions $r \in \{1, 4, 8, 16, 24\}$ and window size $W \in \{50, 100, 200, 400, 1000, 2000, 4000\}$. We use the same values for m as with KRT. More repetitions and larger windows lead to more accurate routing through more accurate retrieval at the cost of higher running time. The window size can be tuned directly at query time. Similarly, the number of repetitions can be increased incrementally when needed, without having to rebuild earlier SortingLSH indices. The results in Figure 3 use the best performing configuration with search budget $b = r \cdot 2W \leq 50K$. We use SimHash as the locality sensitive hash function with 64 projections, i.e., the compound hash of each vector contains 64 hash bits. This enables efficient encoding into 64 bit integers and allows us to use standard sorting algorithms to construct the SortingLSH index, instead of more costly string sorting algorithms. Since we use SortingLSH [5], we need not tune the number of bits to an appropriate hash bucket size.

In-Shard Search Configuration for Billion-Scale. For in-shard search, we also use degree 32 and ef_construction = 200. For different in-shard search recall trade-offs we again vary the beam-width parameter ef_search $\in \{50, 80, 100, 150, 200, 250, 300, 400, 500\}$.

Configuration for Million-Scale. On SIFT and GloVe we use a smaller router and HNSW graph with KRT. We set the router size $m = 50K$, the search budget $b = 5K$, number of centroids $l = 32$ and cluster size threshold $\lambda = 200$. The HNSW graphs use degree 16 and ef_construction = 200. For routing we use ef_search = 60, whereas for in-shard search we use ef_search = 120. These parameters are reduced compared to billion-scale, to cut down on running time for routing. For GloVe we use distance-based routing. For SIFT1M we use the distance-frequency-hybrid routing.

4.4 Routing: KRT vs HRT

In Figure 3 we compare HRT versus KRT by recall versus number of shards probed η, using GP as the partitioning method and exhaustive search in the shards. The plot also shows the performance of the aforementioned routing oracle, to provide context. While HRT performs decently, empirically KRT consistently finds better

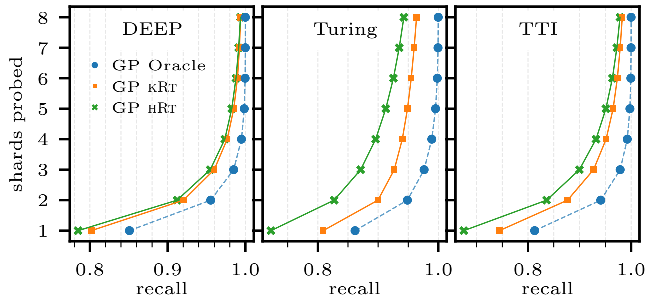

Figure 3: Recall vs shards probed η comparing routing with HRT vs KRT using GP as the partition and exhaustive search in the shards.

routes. This is caused by the uniform random sampling which is necessary for the theoretical analysis of HRT. In the remainder of the evaluation we therefore focus on KRT exclusively. We note that the relevance of HRT lies in its theoretical guarantees.

4.5 Large-Scale Throughput Evaluation

Next, we perform an evaluation of recall vs throughput (queries per second) of approximate nearest neighbor queries. Due to resource constraints we simulate distributed execution on a single machine, processing hosts one after another. While this setup does not take into account networking latency, we note that its impact should be negligible: communication latency in modern networks is less than 1 microsecond [39] whereas HNSW search latency is around 1 millisecond or higher. Moreover, our algorithms (and all sharding-based algorithms in general) perform the least amount of communication possible – only the query vector as well as neighbor IDs and distances are transmitted.

We assume a distributed architecture where each machine hosts one shard and the routing index. For routing, queries are distributed evenly. The machine that receives a query forwards it to the hosts that are supposed to be probed.

We use HNSW to search in the shards and to accelerate routing on the KRT points. In total we use 60 hosts. This is larger than the number of shards, as we replicate popular shards to counteract query load imbalance. We process queries on 32 cores per host in parallel. Throughput is calculated based on the maximum runtime of any host. The work for query routing is distributed evenly across hosts, whereas in-shard searches are only accounted for the queries that probe the shard on the host. Replicas of the same shard evenly distribute work amongst themselves.

To obtain different throughput-recall trade-offs, we try different configurations of KRT (index size m) and HNSW (ef_search), as well as number of shards probed η or the probe filter methods proposed by [13] and [8], which determine a different η for each query. For Pyramid, we also included its native routing method. Additionally, for k-means based partitioning methods (KM, OKM, BKM, OBKM), we included their native routing method with one center per shard. We only plot the Pareto-optimal configurations, which is standard practice [16]. We note that in order to obtain the most competitive configurations, it is necessary to tune all of the above parameters.

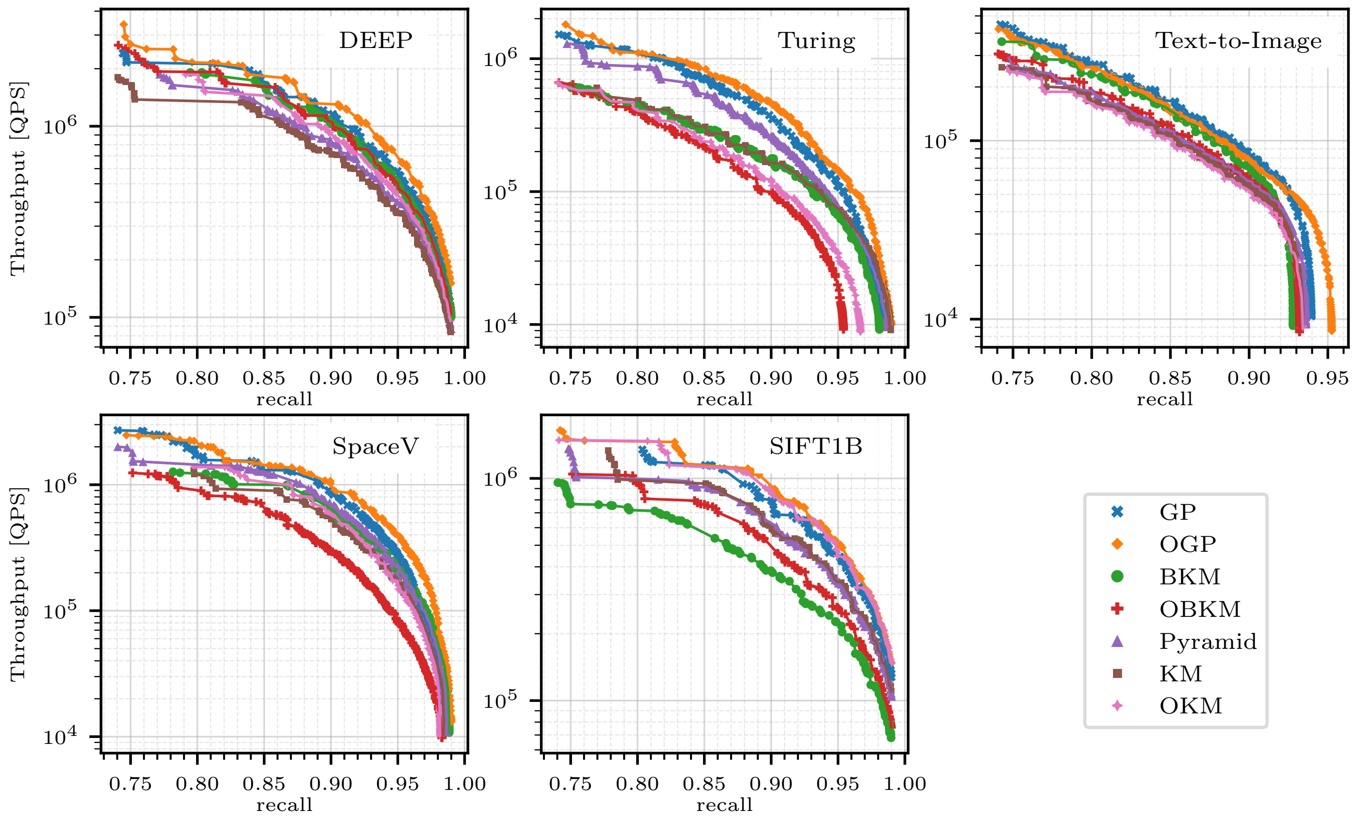

Figure 4: Throughput vs recall evaluation on big-ann-benchmarks for $k = 10$ nearest neighbors, comparing both overlapping and disjoint partitions. Using KRT with HNSW acceleration for routing and HNSW search in the shards. For Pyramid and k-means based partitionings we additionally use their native routing methods.

Table 3: Throughput in 10^3 queries per second at 0.9 recall for $k = 10$ nearest neighbors. Best results highlighted in bold.

		$k = 10$			
QPS [$\cdot 10^3$]	GP	OGP	BKM	OBKM	Pyramid
DEEP	1157.7	**1299.3**	1030.2	1056.8	810.9
Turing	353.3	**450.4**	177.5	99.2	261.5
TTI	**81.6**	76.4	70.6	59.6	59.4
SpaceV	872.3	**893.7**	608.2	292.9	689.1
SIFT1B	793.8	**895.8**	376.4	459.9	608.7

Table 4: Partitioning times in minutes. GB = graph-building. For graph partitioning we report the time for KaMinPar as GP and the time for GB + GP in brackets

	GB	GP	(GB+GP)	OGP	BKM	OBKM	Pyramid
DEEP	63	24	(87)	+10	24	+16	32
Turing	69	17	(86)	+11	36	+18	35
TTI	107	17	(124)	+10	65	+34	60
SpaceV	75	15	(90)	+8	56	+16	34
SIFT1B	89	20	(109)	+8	823	+19	41

The Pareto optimal configurations combine high-quality routing with low-quality shard search and vice versa across the Pareto front.

Figure 4 shows the recall vs throughput plot for $k = 10$ nearest neighbors. In the full version of the paper [21], we also report results for $k = 1$ and $k = 100$ nearest neighbors. The overall trends are the same as for $k = 10$. The algorithm ranking remains the same. Throughput is higher for $k = 1$ and lower for $k = 100$. Additionally, Table 3 shows the throughput for a fixed recall value of 0.9 for $k = 10$. GP outperforms all baselines, and does so by a large margin on Turing and TTI. Adding overlap with OGP improves results further across the whole Pareto front on DEEP, Turing, SpaceV and SIFT1B. On Text-to-Image, OGP loses to GP on recall below 0.93, but OGP can achieve higher maximum recall. Note that the maximum recall is constrained by the approximate in-shard HNSW search. Overall GP or OGP improves QPS over the next best competitor at 90% recall for $k = 10$ by 1.23x, 1.72x, 1.16x, 1.3x, and 1.03x respectively on DEEP, Turing, TTI, SpaceV and SIFT1B, as well as 1.27x in the geometric mean. Moreover OGP improves over GP by up to 1.27x and by 1.09x in the geometric mean.

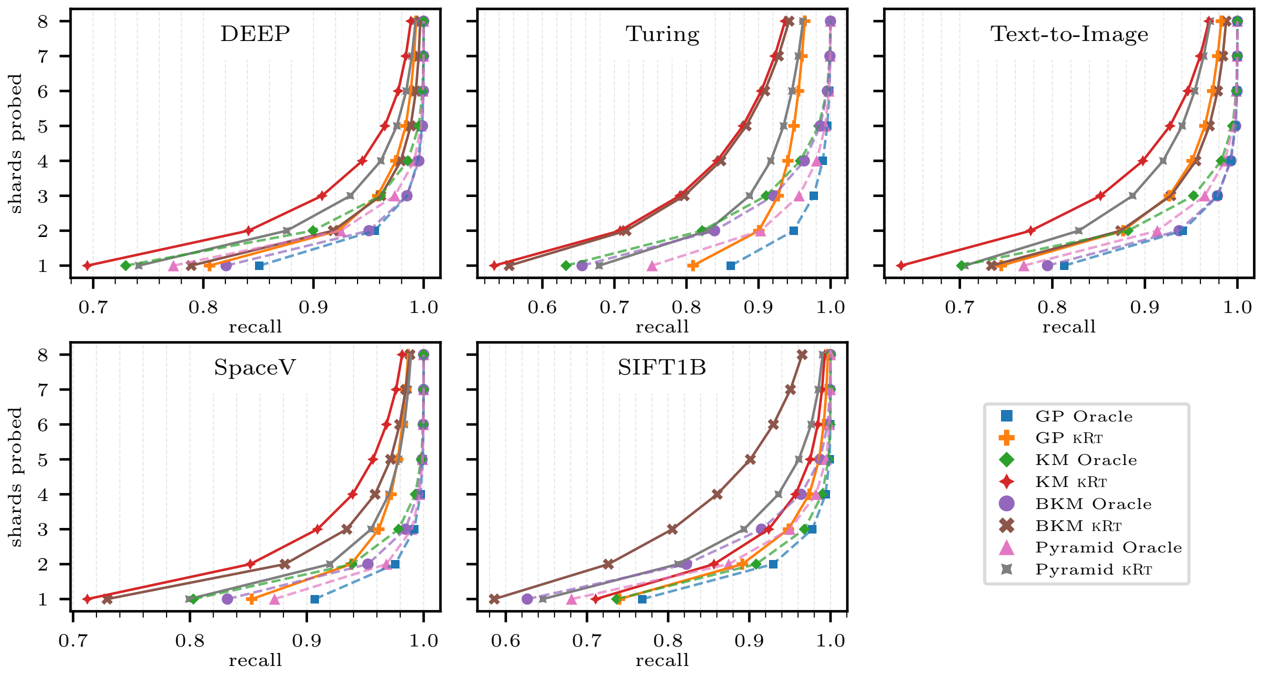

Figure 5: Evaluating disjoint partitioning methods with κRt and routing oracle (dashed), with exhaustive search in the shards for $k = 10$ nearest neighbors.

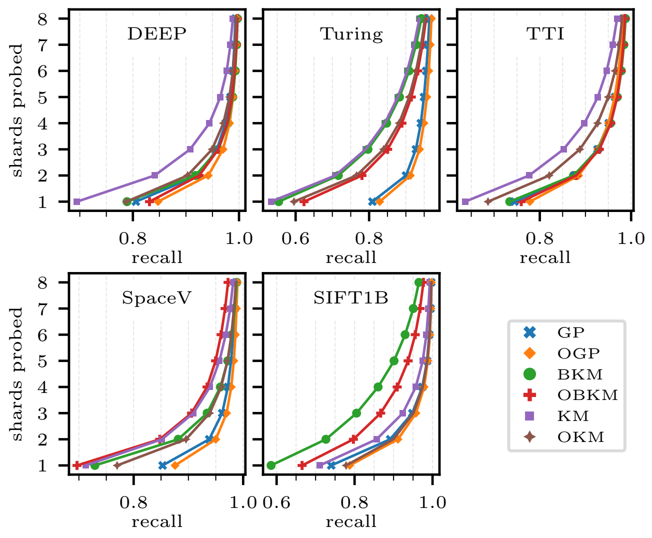

Figure 6: Evaluating overlapping vs disjoint partitions with κRt routing and exhaustive shard search for $k = 10$ nearest neighbors.

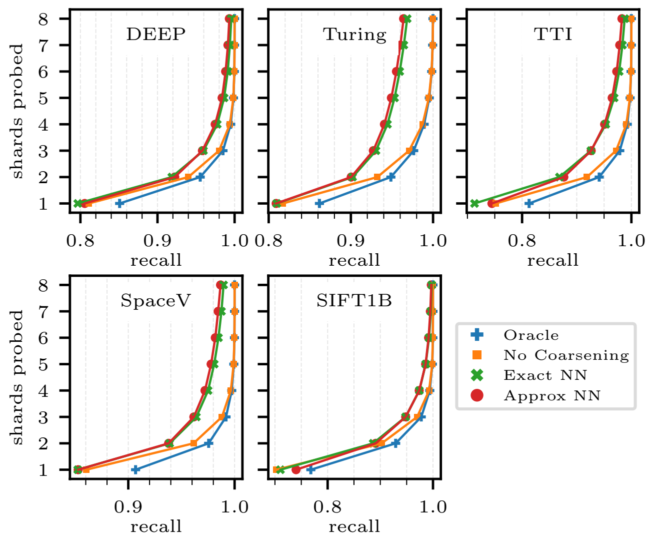

Figure 7: Ablation study analyzing the sources of routing losses for $k = 10$ nearest neighbors. Different curves correspond to different variants of a routing algorithm used. We note that all but *Approx NN* are hypothetical and/or infeasible in practice. Also note that we used distance-based routing on TTI to enable comparison with *Exact NN*, whereas in other experiments we use the frequency-distance hybrid on TTI.

4.6 Training Time

In Table 4 we report partitioning times. The improved query performance of GP does come at the cost of a moderately increased partitioning time compared to the baselines. This is due to the graph-building step (GB). While previous works [15, 17, 25] identified graph partitioning as the bottleneck, we observed that it can be made very fast by using the KaMinPar [23] partitioner. The overall partitioning times are quite moderate for datasets of this scale, and

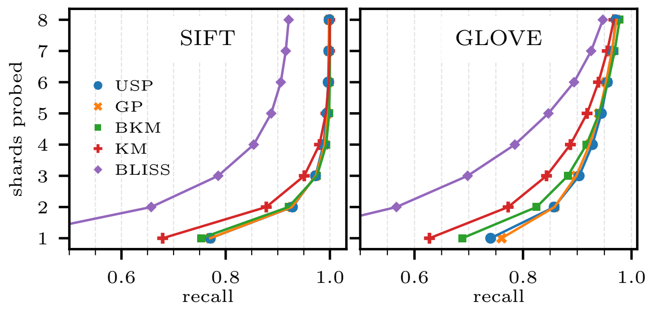

Figure 8: Comparison against neural network baselines.

our implementations of the baselines are competitive. The Pyramid paper reports a partitioning time of 87 minutes on a 500M sample of DEEP, demonstrating that our reimplementation is faster. Reference [9] reports 4-5 days training time for their hierarchical balanced k-means tree SPANN on SpaceV and SIFT. The long training time of BKM on SIFT1B is caused by the need to iteratively escalate to a large penalty factor to achieve a balanced solution.

While our graph-building implementation is already optimized, it can be further accelerated using GPUs [32] or more hardware-friendly implementation of bulk distance computations [24].

Coarsening with кRт takes roughly 1000s on Turing, 800s on DEEP, and 2000s on TTI. HNSW training in a shard of roughly 25M points takes 600s-1500s for Turing and DEEP, and 800s-1900s for TTI. Note that these numbers apply to all baseline partitioners, as they use the same routing algorithms and in-shard search. Training нRт takes under 20 seconds.

4.7 Analyzing Partitioning and Routing Quality

To further assess the quality of the partitions and routing, we study recall independent of HNSW searches in the shards. Specifically, we look at the number of probed shards versus the recall with exhaustive search in the shards, in order to assess the quality of the routing algorithm. All recall values reported in Figures 5 to Figure 7 and Section 4.7 assume exhaustive search. Furthermore, to assess the quality of the partition alone, we look at recall with a *hypothetical* routing oracle that knows the entire dataset and picks an optimal sequence of shards to probe.

Disjoint Partitions. Figure 5 shows the results for disjoint partitions with the oracle marked as dashed lines. GP outperforms Pyramid, KM and BKM on all three datasets. In particular on Turing the margin of improvement is very significant: +25% recall over BKM and +13% over Pyramid at $\eta = 1$ probes with кRт and +11%, respectively +20% with the oracle.

Overlapping Partitions. Next, we investigate the effects of using 20% overlap in Figure 6. Since we use more shards instead of increasing their size ($s' = 48$ instead of $s = 40$) , it is not guaranteed that overlap will lead to strictly better results. Fortunately, we see consistent and substantial improvement of all overlapping methods over their disjoint counterparts. For example, on Text-to-Image, OGP achieves 77.8% 10-recall in the first shard (85.5% routing oracle) compared to GP with just 74.4% (81.2% oracle). The improvements are most pronounced when the disjoint partitioning method is still performing poorly, as there is more headroom. For example on DEEP, the 64% recall for KM gets boosted to 74.4% with OKM. Overall these results demonstrate that overlapping shards significantly boost recall. We note that while this was known for KM [8, 45], we are the first to show this for GP.

Table 5: Routing time in microseconds per query for different batch sizes b on SIFT and GLOVE. Queries in a batch are processed in parallel.

	SIFT			GLOVE		
	$b = 1$	$b = 32$	$b = \infty$	$b = 1$	$b = 32$	$b = \infty$
кRт	95	13	2.4	92	11	2.1
кRт + HNSW	110	37	7.8	106	33	7.6
BLISS	660	150	110	730	110	110
USP	320	20	1.2	512	23	1.2

Losses. While our routing algorithms perform well, there is still a significant gap to the optimal routing oracle. We leverage fast inexact components in two places. We coarsen the pointset, and we use an approximate search index to speed up the routing ANN query. In the following, we thus investigate how much the inexact components contribute to the losses by replacing them with an exact version. We present the results in Figure 7.

We test distance-based routing based on the entire pointset to determine the effect of coarsening and study how distance-based routing loses compared to the oracle (No Coarsening). Additionally, we test computing distances to all points in R (Exact NN), to detect how much approximate search loses in routing by missing relevant points as employed in the base version of the algorithm (Approx NN). Note that for all five datasets including Text-to-Image, we use pure distance-based routing, not the frequency-distance-hybrid.

At $\eta = 1$ probed shard, routing by ranking distances already loses significantly – see Oracle against No Coarsening. It is able to catch up at $\eta = 3$, suggesting that the distinction among the top 3 should be improved, but overall ranking distances is the correct approach. At $\eta > 1$ coarsening incurs the highest losses – see No Coarsening vs Exact NN. Fortunately, Approx NN incurs only a small loss against Exact NN; at $\eta \geq 3$ on Turing and $\eta \geq 5$ on TTI.

4.8 Small-Scale Evaluation for Learned Routing

In this section, we compare with two additional baselines BLISS [25] and USP [17], which are examples of neural routing methods. Training BLISS and USP is too slow to run on the big-ann datasets, so we test on SIFT1M and GLOVE, which have roughly a million points, partitioned into $s = 16$ shards. Queries are run sequentially, except for the batch-parallel results in Table 5 which use 64 cores. Index training uses 128 cores. The results for GP and (B)KM use кRт routing without HNSW acceleration.

Figure 8 shows the recall versus number of probed shards. First, we observe that GP and USP perform similarly and are the frontrunners on both datasetes, whereas BLISS is completely outperformed. This comparison uses exhaustive search in the shards, which takes 1.98ms per shard probe on SIFT. Alternatively, using HNSW takes 0.23ms per shard probe with near-equivalent recall (77% vs 76.7% in the first shard). BLISS exhibits 13-15ms for exhaustive shard search, which is presumably an implementation issue of interfacing from Python to C++. For USP we cannot get a clean measurement, because the shard-search is mixed with the recall calculation. In Table 5 we report the routing times. USP and BLISS benefit from routing in batches, as PyTorch leverages parallelism and batched linear algebra operations. Our approach similarly benefits from parallelism, but could be further improved with batched distance computations. In contrast to the results on large-scale data, we observe that routing with HNSW is slower than with κRT.

In terms of retrieval performance, USP is a good data structure. However, it is extremely slow to train, taking 6 hours on 128 cores for GLOVE. The main bottleneck is training the routing model on the base points over several epochs. On the other hand BLISS is fast to train at 70s (neural network) + 566s (k-NN graph construction), but has poor retrieval performance. We excluded Neural LSH [15] from the retrieval comparison as it is already outperformed by USP, but remark that its neural net training took over two days. Our method has the best retrieval speed at relevant batch sizes *and* is also the fastest to train at just 4.76s, of which 2.95s are graph-building, 0.88s partitioning, and 0.93s κRT.

5 RELATED WORK

Approximate k-Nearest Neighbor Graph Construction. There are several works on approximate k-NN graph construction, which are relevant. *NN-Descent* by Dong et al [14] is an iterative improvement algorithm, which refines an initial k-NN graph. This can be a random initial graph or computed with some other method. The intuition underlying their iterative improvement approach is to introduce neighbors of neighbors to one another, as the neighbors of a node's current neighbors are likely good neighbor candidates themselves. One round of NN-Descent is typically implemented by computing the transpose of the current graph and merging the resulting lists. EFANNA [19] feeds the pointset as queries to k-d trees to build an initial k-NN graph to be refined with NN-Descent. Tang et al [46] utilize random projection trees instead of k-d trees to build the initial graph.

Another approach is based on locality sensitive hashing and using all-pairs comparisons to emit edges in the resulting hash buckets [47]. Our graph construction method is an instatiation of this approach with a data-dependent LSH function (the cluster assignment). It is also related to HCNNG [37], a graph-structured ANNS index, which uses two random pivots for splitting the data and computes a degree-constrained MST as the recursive base case to build a navigable search graph.

Graph-Structured Indices for Approximate Nearest Neighbor Search. Due to the curse of dimensionality, and the resulting sub-par performance of data structures like k-d trees on high-dimensional data, recent advances in ANNS have focused on graph-structured indices. DiskANN [44] and HNSW (Hierarchical Navigable Small-World Graph) [35] have become widely established due to their excellent recall-time trade-off. The general idea is to build a navigable proximity graph on the dataset points, such that a beam search converges to nearest neighbors of a query. In each step, the explored vertex closest to the query is expanded by scanning its neighbors in the search graph and computing their distance to the query. Explored vertices are stored in a size-constrained priority queue (the beam) to restrict the search direction towards near neighbors.

Distributed Nearest Neighbor Search. Despite the need for distributed ANNS in scaling beyond a billion points, there is limited published work on the subject [8, 13, 36]. We believe this is partially due to the difficulty of query load imbalance, as also briefly noted in SPANN [8]. Their approach is to load-balance queries during the partitioning step by incorporating query access frequencies. However, if the online query distribution differs or the training set is not sufficiently representative, this can still incur load imbalance. We argue that in a large-scale setup, where replica machines are needed for fault-tolerance and increased throughput, it is worthwhile to selectively replicate heavily loaded machines, which can also be made robust to distribution shift by tracking query access frequencies during the online stage.

FLANN [36] is a distributed search solution, which uses random sharding, routes a query to all machines, and uses a k-means tree as the in-memory index for each shard. Note that the QPS increase from adding replicas is much more limited for random sharding, because a query is always routed to all shards, as opposed to locality-optimized partitions (such as GP or BKM) where only a small subset of the shards is probed.

Graph Partitioning. The KaMinPar [23] graph partitioner follows the multilevel approach [27], where we first repeatedly contract vertex clusters to obtain successively smaller graphs, which constitute the levels. On the smallest graph, we compute an initial partition using a portfolio of randomized greedy heuristics [22]. In the uncoarsening stage we then revert the contractions level by level, project the partition to the next graph and iteratively improve it using refinement algorithms. KaMinPar uses size-constrained label propagation [40] for coarsening and refinement, a simple iterative and highly parallel greedy vertex moving procedure.

6 CONCLUSION

We presented fast, modular and high-quality routing methods which unleash balanced graph partitioning for large-scale nearest neighbor search and establish it as the partitioning method of choice. Our routing methods build upong center-based routing, and achieve compatibility with graph partitioning by training representatives independently for each shard. Additionally, employing multiple and diverse representative vectors to encapsulate the hierarchical substructure within shards substantially improves routing decisions and thus recall. Moreover, our overlapping partitioning method, based on further eliminating k-NN graph cut edges, substantially improves the recall-throughput trade-off compared to the disjoint partition. Our benchmarks on billion-scale, high-dimensional data show that our methods achieve up to 1.72x higher throughput at 90% recall than the best baseline, and 1.27x in the geometric mean.

As a future work it would be interesting to explore accuracy and efficiency improvements for routing, for example exploring quantization to compress the routing index. Another direction is to study the benefits of our approach for partitioning ANNS problems across different types of computing units, e.g. GPUs. Additionally, we are interested in advanced partitioning cost functions that further optimize locality in approximate nearest neighbor search.

REFERENCES

[1] Alexandr Andoni, Piotr Indyk, and Ilya Razenshteyn. 2018. Approximate nearest neighbor search in high dimensions. In *Proceedings of the International Congress of Mathematicians: Rio de Janeiro 2018*. World Scientific, 3287–3318.

[2] Alexandr Andoni, Thijs Laarhoven, Ilya Razenshteyn, and Erik Waingarten. 2017. Optimal hashing-based time-space trade-offs for approximate near neighbors. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 47–66.

[3] Artem Babenko and Victor Lempitsky. 2016. Efficient indexing of billion-scale datasets of deep descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2055–2063.

[4] Dmitry Baranchuk, Artem Babenko, and Yury Malkov. 2018. Revisiting the Inverted Indices for Billion-Scale Approximate Nearest Neighbors. In *Computer Vision - ECCV 2018 (Lecture Notes in Computer Science)*, Vol. 11216. Springer, 209–224. https://doi.org/10.1007/978-3-030-01258-8_13

[5] Mayank Bawa, Tyson Condie, and Prasanna Ganesan. 2005. LSH Forest: Self-tuning Indexes for Similarity Search. In *Proceedings of the 14th international conference on World Wide Web, WWW 2005*, Allan Ellis and Tatsuya Hagino (Eds.). ACM, 651–660. https://doi.org/10.1145/1060745.1060840

[6] Sebastian Bruch. 2024. Foundations of Vector Retrieval. *arXiv preprint arXiv:2401.09350* (2024).

[7] Ümit V. Çatalyürek, Karen D. Devine, Marcelo Fonseca Faraj, Lars Gottesbüren, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Dorothea Wagner. 2023. More Recent Advances in (Hyper)Graph Partitioning. *Comput. Surveys* 55, 12 (2023), 253:1–253:38. https://doi.org/10.1145/3571808

[8] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. SPANN: Highly-efficient Billion-scale Approximate Nearest Neighborhood Search. In *Advances in Neural Information Processing Systems 34: NeurIPS 2021*, Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan (Eds.). 5199–5212. https://proceedings.neurips.cc/paper/2021/hash/299dc35e747eb77177d9cea10a802da2-\Abstract.html

[9] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. SPANN Paper Discussion. https://openreview.net/forum?id=-1rrzmJCp4&noteId=zhMe9y8w25b. Accessed: 2023-04-14.

[10] Xi Chen, Rajesh Jayaram, Amit Levi, and Erik Waingarten. 2022. New streaming algorithms for high dimensional EMD and MST. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*. 222–233.

[11] Rieke de Maeyer, Sami Sieranoja, and Pasi Fränti. 2023. Balanced k-means Revisited. *Applied Computing and Intelligence* 3, 2 (2023), 145–179.

[12] Shiyuan Deng, Xiao Yan, Kelvin Kai Wing Ng, Chenyu Jiang, and James Cheng. 2019. Pyramid: A General Framework for Distributed Similarity Search. *CoRR* abs/1906.10602 (2019). arXiv:1906.10602 http://arxiv.org/abs/1906.10602

[13] Shiyuan Deng, Xiao Yan, Kelvin Kai Wing Ng, Chenyu Jiang, and James Cheng. 2019. Pyramid: A General Framework for Distributed Similarity Search on Large-scale Datasets. In *2019 IEEE International Conference on Big Data (IEEE BigData)*, Chaitanya K. Baru, Jun Huan, Latifur Khan, Xiaohua Hu, Ronay Ak, Yuanyuan Tian, Roger S. Barga, Carlo Zaniolo, Kisung Lee, and Yanfang (Fanny) Ye (Eds.). IEEE. https://doi.org/10.1109/BigData47090.2019.9006219

[14] Wei Dong, Moses Charikar, and Kai Li. 2011. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*. ACM, 577–586. https://doi.org/10.1145/1963405.1963487

[15] Yihe Dong, Piotr Indyk, Ilya P. Razenshteyn, and Tal Wagner. 2020. Learning Space Partitions for Nearest Neighbor Search. In *8th International Conference on Learning Representations, ICLR 2020*. OpenReview.net. https://openreview.net/forum?id=rkenmREFDr

[16] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The Faiss library. *arXiv preprint arXiv:2401.08281* (2024).

[17] Abrar Fahim, Mohammed Eunus Ali, and Muhammad Aamir Cheema. 2022. Unsupervised Space Partitioning for Nearest Neighbor Search. *CoRR* abs/2206.08091 (2022). https://doi.org/10.48550/arXiv.2206.08091 arXiv:2206.08091

[18] Charles M. Fiduccia and Robert M. Mattheyses. 1982. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference, DAC 1982*. ACM/IEEE, 175–181. https://doi.org/10.1145/800263.809204

[19] Cong Fu and Deng Cai. 2016. EFANNA : An Extremely Fast Approximate Nearest Neighbor Search Algorithm Based on kNN Graph. *CoRR* abs/1609.07228 (2016). arXiv:1609.07228 http://arxiv.org/abs/1609.07228

[20] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *Proceedings of the VLDB Endowment* 12, 5 (2019), 461–474. https://doi.org/10.14778/3303753.3303754

[21] Lars Gottesbüren, Laxman Dhulipala, Rajesh Jayaram, and Jakub Lacki. 2024. Unleashing Graph Partitioning for Large-Scale Nearest Neighbor Search. *CoRR* abs/2403.01797 (2024). arXiv:2403.01797 https://doi.org/10.48550/arXiv.2403.01797

[22] Lars Gottesbüren, Tobias Heuer, Nikolai Maas, Peter Sanders, and Sebastian Schlag. 2024. Scalable High-Quality Hypergraph Partitioning. *ACM Trans. Algorithms* 20, 1 (2024), 9:1–9:54. https://doi.org/10.1145/3626527

[23] Lars Gottesbüren, Tobias Heuer, Peter Sanders, Christian Schulz, and Daniel Seemaier. 2021. Deep Multilevel Graph Partitioning. In *29th Annual European Symposium on Algorithms, ESA 2021*. https://doi.org/10.4230/LIPIcs.ESA.2021.48

[24] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating Large-Scale Inference with Anisotropic Vector Quantization. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020 (Proceedings of Machine Learning Research)*, Vol. 119. PMLR, 3887–3896. http://proceedings.mlr.press/v119/guo20h.html

[25] Gaurav Gupta, Tharun Medini, Anshumali Shrivastava, and Alexander J. Smola. 2022. BLISS: A Billion scale Index using Iterative Re-partitioning. In *KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, Aidong Zhang and Huzefa Rangwala (Eds.). ACM, 486–495. https://doi.org/10.1145/3534678.3539414

[26] Ben Harwood and Tom Drummond. 2016. FANNG: Fast approximate nearest neighbour graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 5713–5722.

[27] Bruce Hendrickson and Robert Leland. 1993. *A Multilevel Algorithm for Partitioning Graphs*. Technical Report SAND93-1301. Sandia National Laboratories.

[28] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. 604–613.

[29] Masajiro Iwasaki and Daisuke Miyazaki. 2018. Optimization of Indexing Based on k-Nearest Neighbor Graph for Proximity Search in High-dimensional Data. *CoRR* abs/1810.07355 (2018). arXiv:1810.07355 http://arxiv.org/abs/1810.07355

[30] Shikhar Jaiswal, Ravishankar Krishnaswamy, Ankit Garg, Harsha Vardhan Simhadri, and Sheshansh Agrawal. 2022. OOD-DiskANN: Efficient and Scalable Graph ANNS for Out-of-Distribution Queries. *CoRR* abs/2211.12850 (2022). https://doi.org/10.48550/arXiv.2211.12850 arXiv:2211.12850

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

[32] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2021), 535–547. https://doi.org/10.1109/TBDATA.2019.2921572

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

[34] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2007. Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB) 2007*. ACM, 950–961. http://www.vldb.org/conf/2007/papers/research/p950-lv.pdf

[35] Yury A. Malkov and Dmitry A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42, 4 (2020), 824–836. https://doi.org/10.1109/TPAMI.2018.2889473

[36] Marius Muja and David G. Lowe. 2014. Scalable Nearest Neighbor Algorithms for High Dimensional Data. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36, 11 (2014), 2227–2240. https://doi.org/10.1109/TPAMI.2014.2321376

[37] Javier Alvaro Vargas Muñoz, Marcos André Gonçalves, Zanoni Dias, and Ricardo da Silva Torres. 2019. Hierarchical Clustering-Based Graphs for Large Scale Approximate Nearest Neighbor Search. *Pattern Recognition* 96 (2019). https://doi.org/10.1016/j.patcog.2019.106970

[38] David Nistér and Henrik Stewénius. 2006. Scalable Recognition with a Vocabulary Tree. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2006)*. IEEE Computer Society, 2161–2168. https://doi.org/10.1109/CVPR.2006.264

[39] Oded Paz. 2014. InfiniBand Essentials Every HPC Expert Must Know. (2014). https://www.hpcadvisorycouncil.com/events/2014/swiss-workshop/presos/Day_1/1_Mellanox.pdf HPC Advisory Council Switzerland Conference.

[40] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. 2007. Near Linear Time Algorithm to Detect Community Structures in Large-Scale Networks. *Physical Review E* 76, 3 (2007), 036106.

[41] Stuart J. Russell and Peter Norvig. 2009. *Artificial Intelligence: a modern approach* (3 ed.). Pearson.

[42] Gregory Shakhnarovich, Trevor Darrell, and Piotr Indyk. 2008. Nearest-neighbor methods in learning and vision. *IEEE Trans. Neural Networks* 19, 2 (2008), 377.

[43] Harsha Vardhan Simhadri, George Williams, Martin Aumüller, Matthijs Douze, Artem Babenko, Dmitry Baranchuk, Qi Chen, Lucas Hosseini, Ravishankar Krishnaswamy, Gopal Srinivasa, Suhas Jayaram Subramanya, and Jingdong Wang.

2021. Results of the NeurIPS'21 Challenge on Billion-Scale Approximate Nearest Neighbor Search. In *NeurIPS 2021 Competitions and Demonstrations Track (Proceedings of Machine Learning Research)*, Douwe Kiela, Marco Ciccone, and Barbara Caputo (Eds.), Vol. 176. PMLR, 177–189. https://proceedings.mlr.press/v176/simhadri22a.html

[44] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnaswamy, and Rohan Kadekodi. 2019. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 13748–13758. https://proceedings.neurips.cc/paper/2019/hash/09853c7fb1d3f8ee67a61b6bf4a7f8e6-\Abstract.html

[45] Philip Sun, David Simcha, Dave Dopson, Ruiqi Guo, and Sanjiv Kumar. 2023. SOAR: Improved Indexing for Approximate Nearest Neighbor Search. In *Thirty-seventh Conference on Neural Information Processing Systems*.

[46] Jian Tang, Jingzhou Liu, Ming Zhang, and Qiaozhu Mei. 2016. Visualizing Large-scale and High-dimensional Data. In *Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, April 11 - 15, 2016*. ACM, 287–297. https://doi.org/10.1145/2872427.2883041

[47] Yan-Ming Zhang, Kaizhu Huang, Guanggang Geng, and Cheng-Lin Liu. 2013. Fast kNN Graph Construction with Locality Sensitive Hashing. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2013, Prague, Czech Republic, September 23-27, 2013, Proceedings, Part II (Lecture Notes in Computer Science)*, Vol. 8189. Springer, 660–674. https://doi.org/10.1007/978-3-642-40991-2_42