MIRAGE-ANNS: Mixed Approach Graph-based Indexing for Approximate Nearest Neighbour Search

by

Sairaj Voruganti

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2025

© Sairaj Voruganti 2025

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Approximate nearest neighbour search (ANNS) on high dimensional vectors is important for numerous applications, such as search engines, recommendation systems, and more recently, large language models (LLMs), where Retrieval Augmented Generation (RAG) is used to add context to an LLM query. Graph-based indexes built on these vectors have been shown to perform best but have challenges. These indexes can either employ refinement-based construction strategies such as K-Graph and NSG, or increment-based strategies such as HNSW. Refinement-based approaches have fast construction times, but worse search performance and do not allow for incremental inserts, requiring a full reconstruction each time new vectors are added to the index. Increment-based approaches have good search performance and allow for incremental inserts, but suffer from slow construction. This work presents `MIRAGE-ANNS` (**M**ixed **I**ncremental **R**efinement **A**pproach **G**raph-based **E**xploration for Approximate Nearest neighbour Search) that constructs the index as fast as refinement-based approaches while retaining search performance comparable or better than increment-based ones. It also allows incremental inserts. We show that MIRAGE achieves state of the art construction and query performance, outperforming existing methods by up to 2× query throughput on real-world datasets.

Acknowledgements

Firstly, I would like to thank my supervisor, Dr. Tamer Özsu. I cannot repay him for all the guidance and support he has given me throughout my masters. He took a chance on me and for that I will forever be grateful. His passion for research is contagious and helped motivate me to keep a positive attitude no matter what and strive to put my best work forward.

I would like to thank Professors Khuzaima Daudjee and Jimmy Lin for being on my thesis committee.

I would like to thank my friends in the DSG lab: Kerem, Amin, Gaurav, Anurag. Without all of the discussions in lab and their guidance and support during the hard times I would not have been able to finish my work. When I first moved to Waterloo I did not know anyone or much about research, and Kerem showed me everything, for that I will always be grateful. Chatting with Gaurav about research and vector indexes helped spark the ideas for what this thesis would become. Working with Amin on a course project on this topic allowed me to learn it much more in depth, and going to the gym with him helped me get my mind off my research when I needed to. Seeing Anurag in the lab working consistently inspired me to put more effort into my own research during the times when I felt down.

I would like to thank Aayush and Rishabh, two of my closest friends, for allowing me to come visit them whenever I needed to.

I want to thank Navya, who supported me wholeheartedly throughout my masters, listening to my troubles and studying alongside me.

I would like to thank my uncles and aunts, Pavan Babaya, Sriusha Pinni, Murthy Mavayya, and Sunita Atta. The support and guidance from all of them cannot be overstated. I want to thank my cousins, Saishyam, Arya, Ananya, and Sachin for their support and love. When this work got accepted at SigMod 2025, they were maybe more excited than I was.

I want to thank my grandparents, Namana, Ammama, and Tata for their prayers and love during my whole life and specifically during my masters.

Lastly, I would like to thank my parents Sundari and Kaladhar. Both my parents are distinguished researchers themselves, but they allowed me to make my own mistakes, always supporting me along the way and giving me the belief that I could do anything I put my mind to. This masters is dedicated to them.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements iv

List of Figures vii

List of Tables ix

1 Introduction 1

2 Background and Related Work 4

2.1 Basic Definitions 4

2.2 Vector Indexes 5

2.3 Graph-based Indexes and ANNS 8

2.4 Refinement-based Graph Construction 12

2.5 Increment-based Graph Construction 14

2.6 Other Related Works 17

3 MIRAGE Index 19

3.1 Motivation and Intuition 19

3.2 Graph Construction 20
3.3 Search 22
3.4 Handling Updates 24
3.5 Complexity of Algorithms 26

4 Evaluation **27**
4.1 Experimental Setup 27
4.2 Memory Cost 29
4.3 Index Construction Performance 31
4.4 Search Performance 32
4.5 Graph Qualities 43
4.6 Parameter Experiments 43
4.7 Ablation Study 46
4.8 Discussion 49

5 Conclusion **51**

References **53**

List of Figures

2.1 Locality Sensitive Hashing 6

2.2 Hierarchical balanced clustering of SPANN [13]. The vectors in the large (yellow) cluster are iteratively balanced and partitioned into a few smaller (green) clusters, continuing this process until each resulting cluster contains only a limited number of vectors (blue clusters). 8

2.3 Product Quantization 9

2.4 Original Dataset 11

2.5 Graph-based index performing ANNS to find the closest neighbour to the query 11

2.6 RNG rule 12

2.7 Refinement-based construction 12

2.8 Simple Illustration of NN-Descent join. 14

2.9 Increment-based construction 15

2.10 Illustration of HNSW. The search path is shown by red arrows with a fixed entry points and the query point is in green. 16

3.1 MIRAGE Construction 20

3.2 Diagram of the MIRAGE index. The search path is shown by red arrows with a fixed entry points and the query point is in green. Due to Layer 0 being constructed with a refinement-based approach, MIRAGE is much faster than HNSW in terms of indexing speed. MIRAGE also has a sparser bottom layer which allows for it's search efficiency to be better than HNSW, as much as 2x. 25

4.1 Comparison of peak memory usage for different methods 30
4.2 Indexing time of different graph-based methods. We cap the y-axis at 1000s for readability reasons. For the Tiny5M dataset, HNSW takes 1856.485s and for Sift10M HNSW takes 1051.475s and LSH-APG takes 1249.135s . . 31
4.3 QPS (log scale) vs recall where $k = 1$. 33
4.4 QPS (log scale) vs recall where $k = 1$. 34
4.5 QPS (log scale) vs recall where $k = 10$. 35
4.6 QPS (log scale) vs recall where $k = 10$. 36
4.7 QPS (log scale) vs recall where $k = 20$. 37
4.8 QPS (log scale) vs recall where $k = 20$. 38
4.9 QPS (log scale) vs recall where $k = 100$. 39
4.10 QPS (log scale) vs recall where $k = 100$. 40
4.11 QPS (log scale) vs ADR of the Glove and Crawl datasets where $k = 1$ and $k = 10$. 41
4.12 QPS (log scale) vs ADR of the Glove and Crawl datasets where $k = 20$ and $k = 100$. 42
4.13 Effect of different S values on construction time and search performance (QPS vs ADR) 45
4.14 Effect of different R values on construction time and search performance (QPS vs ADR) 47
4.15 Effect of different Iter values on construction time and search performance (QPS vs ADR) 48
4.16 Ablation Study on MIRAGE Layer 0 vs Hierarchical MIRAGE . Gist dataset is shown on top, Paper dataset is shown on the bottom 50

List of Tables

2.1 Summary of Notation 5

4.1 Summary of datasets with dimensions, base size, query size, LID, and data type. LID [29] denotes how many "degrees of freedom" are necessary to describe the dataset and is used to measure the difficulty of answering NN queries over it; higher LID means harder dataset. 28

4.2 Average vertex out-degree (AOD) and Index size (IS) for each method. Index size is represented in GB. 44

4.3 Number of distance computations to achieve 0.99 recall with $k = 100$. Number in parenthesis represents the difference between HNSW and MIRAGE. 49

Chapter 1

Introduction

Advancements in deep learning have allowed multiple modalities of data, such as images and text, to be leveraged in applications. The multi-modal data is mapped into dense vectors in a vector space [5], and the models are trained in such a way that the "distance" between two vectors represents the similarity between the objects themselves. With the growing popularity of vector embeddings, there has been a surge in vector database offerings, such as Milvus [60], Pinecone [8], and Vespa [9]. Traditional database systems have also incorporated vector data [6]. Similarity search over this vector space is a fundamental operation that allows for more precise and contextualized search over multi-modal data [38]. The search finds the k nearest neighbours of a given point based on the vector distance. Systems integrate vector similarity search into query processing to support workloads that require finding the most similar vectors to a query vector. More recently, similarity search is used in systems supporting vector data in order to perform Retrieval Augmented Generation (RAG), helping to add context to an LLM.

The *k-Nearest neighbour Search* (kNNS) problem is computationally hard to solve in high-dimensional spaces due to the curse of dimensionality [27]. In the worst-case, this means that once the number of dimensions becomes too high, an expensive linear scan is required. Most real-world use cases can afford approximate solutions, leading to the definition of the *approximate nearest neighbour search* (ANNS) problem. ANNS typically relies on an index over the vector dataset. Indexes can be roughly categorized into four types: hashing-based [21], tree-based [13, 12], graph-based [52, 34], and quantization-based [37]. Recent studies [63] have empirically shown that graph-based indexes perform the best, achieving the highest recall and throughput on a variety of datasets.

Graph-based indexes construct a *proximity graph* on the data set, where the vertices

represent the vectors and the edges represent a neighbour relationship between them. The main types of approaches for building graph-based indexes are refinement-based and increment-based [40, 36, 63]. *Refinement-based* approaches start with a random initial graph and then connect each vertex to its k-nearest neighbours, improving the quality of the graph. Examples of this approach include KGraph [24], NSG [18], NN-Descent [24], and RNNDescent [40]. The search performance of refinement-based approaches in terms of recall and QPS (Queries-per-Second) can be low without additional processing post-construction, and they do not allow for incremental inserts, meaning that inserting a new vector requires a full index reconstruction.

Increment-based approaches, on the other hand, traditionally start with an empty graph and add points one at-a-time. Each new insertion is treated as a query point and a nearest neighbour search is performed on the current graph to find its k nearest neighbours for edge connections. The most popular example of an increment-based approach is HNSW [34], which is used by many commercial offerings due to its superior search performance. HNSW constructs a hierarchical graph where the bottom layer contains all the points in the dataset, while each layer above contains increasingly fewer points. The increment-based approaches have good search performance and allow incremental insertions, but have quite high index construction time due to performing nearest neighbour search on the whole graph for every new point.

Thus, there is a trade-off between refinement-based indexes with fast index construction times but inferior search quality and increment-based indexes with slow index construction times and superior search quality [43]. For example, in our experiments on the Tiny5m [1] dataset, RNNDescent can construct its graph around 8× faster than HNSW; however, it achieves around 50% of HNSW QPS during search ($k = 100$).

In this thesis, we introduce MIRAGE, a new graph-based index that tackles the tradeoff between index construction time and query performance: it has a fast index construction time as well as high search performance.

This thesis makes the following contributions:

- We propose MIRAGE, a new approach to constructing graph-based indexes using refinement-based construction to create the bottom layer of the graph, and then building layers in an increment-based manner, with upper layers containing fewer points and longer links, and lower layers containing more points and shorter links. The search uses a greedy algorithm starting from the highest layer and continuing down to the lowest layer. MIRAGE allows for incremental inserts without a full graph reconstruction. It is built on top of the FAISS library [16], a popular open-source library for indexing and performing ANNS.

- We evaluate MIRAGE on real-world datasets of different difficulties as measured by local intrinsic dimensionality (LID) [29], showing that it constructs the index as much as 6× faster than HNSW and 7× faster than NSG, current state of the art approaches, while also improving search efficiency by as much as 2× over the said approaches, where efficiency is measured by QPS (Queries-per-Second) at a fixed recall.
- Through ablation studies and studies on the properties of MIRAGE and other graph-based indexes, we highlight the properties that impact graph-based index performance, and how they can be combined in novel ways to achieve superior performance.

The rest of this thesis is structured as follows: Chapter 2 introduces basic definitions and background information and related work on vector indexes. In Chapter 3 we introduce MIRAGE. In Chapter 4 we present our experiments comparing MIRAGE to existing methods and our ablation studies. Finally, we conclude in Chapter 5 and discuss future directions.

Chapter 2

Background and Related Work

2.1 Basic Definitions

Definition 1. (**Vector dataset**) A *vector dataset* $P = \{p_0, p_1, \ldots, p_{n-1}\}$ has each element p_i as a vector $\mathbf{x^i} = [x_0^i, x_1^i, \ldots, x_{d-1}^i]$ of d dimensions. When clear from the context, the superscript is omitted for clarity.

Definition 2. (**Distance**) The distance between two vectors $\mathbf{x}, \mathbf{y}$ in P, $\delta(\mathbf{x}, \mathbf{y})$ can be computed by several distance functions. The most commonly used distance function is the Euclidean distance (also called l_2 norm).

Definition 3. (**kNNS and kANNS**) Given a vector set P and a query point q, *k-nearest neighbour search* (kNNS(P, q, k)) identifies a set of k vectors $K \subseteq P$ with the smallest distance to q, i.e. $\max_{p \in K} |p, q| \leq \min_{p \in P \setminus K} |p, q|$. The *$k$-approx NNS* (kANNS$(P, q, k)$) is a relaxation of this problem such that vectors in K approximately satisfy this constraint.

Definition 4. (**Recall**) Given the vector set P and a query point q, let K be the result of kNNS(P, q, k), and K' be a result kANNS(P, q, k) produced by a particular kANNS algorithm A. The *k@k' recall* of A for q is then defined as $\frac{|K \cap K'|}{|K|}$ where $k = |K|$ and $k' = |K'|$.

Definition 5. (**Average Distance Ratio**) Given a set of data vectors P and a query point q, let $d(q, p)$ denote the true distance between q and a data point $p \in P$, and $\hat{d}(q, p)$ be the approximate distance produced by a given algorithm. The *average distance ratio* is defined as:

$$\text{ADR}(q) = \frac{1}{|P|} \sum_{p \in P} \frac{\hat{d}(q, p)}{d(q, p)},$$

where $|P|$ is the number of data points in P. The Average Distance Ratio evaluates the relative error of the algorithm's distance estimations over the dataset.

The notation used in this thesis is summarized in Table 2.1.

Table 2.1: Summary of Notation

Symbol	**Description**
P	Vector dataset consisting of p points
$\mathbf{x^i}$	Vector corresponding to point $p_i \in P$
d	Dimension of each vector $\mathbf{x}$
δ	Distance between two vectors
q	Query point
C	Candidate vertices in graph index – priority queue
K	Closest N points so far to query point q
S	Out-degree limit of the initial random graph
M	Degree bound for vertices in MIRAGE
$iter$	Number of graph refinements in Layer 0 of MIRAGE
R	Number of times reverse edges are added in Layer 0 of MIRAGE
$m_L = 1/\ln M$	Level normalization constant for index
n	Size of dataset

2.2 Vector Indexes

Indexes for vector datasets can be categorized into four classes: graph-based, hashing-based, tree-based, quantization-based. In this thesis we focus on graph-based indexes, which are covered in detail in Sections 2.3 – 2.5.

Hashing methods are a family of algorithms that partition the data space into buckets, such that similar items are more likely to be in the same bucket. One of the most popular techniques in this category is Locality-Sensitive Hashing (LSH) [21]. As shown in Figure 2.1, LSH begins by selecting a family of hash functions that are sensitive to the specific distance metric of the data, ensuring that the probability of collision is higher for similar items. Each data point is then processed through hash functions, and the results are concatenated to form a hash key. This key maps the data point to a specific bucket in a hash table. By constructing multiple such hash tables, the algorithm increases the likelihood that similar items will be grouped together in at least one of the tables.

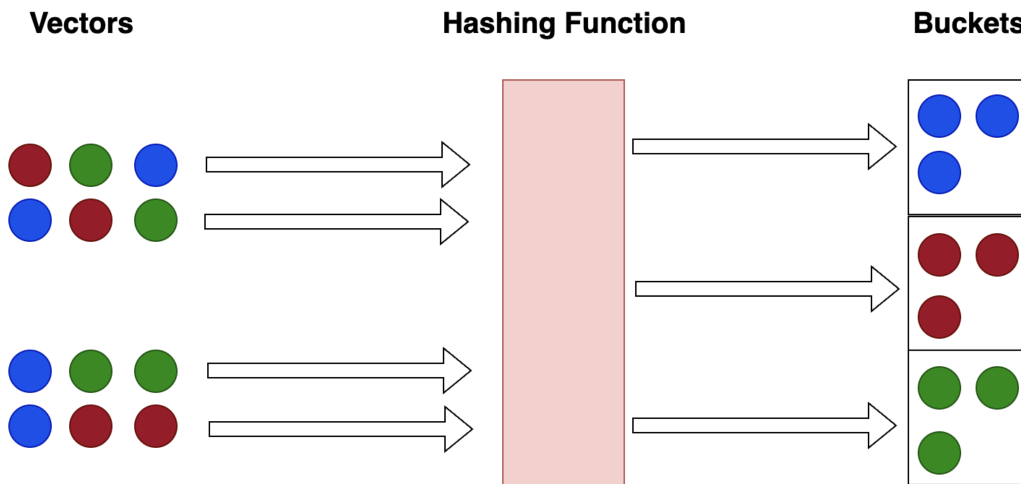


Figure 2.1: Locality Sensitive Hashing

During a query, the algorithm hashes the query point using the same hash functions to identify candidate buckets across all tables. The union of these buckets forms a candidate set, which is then examined to find the approximate nearest neighbours. This method significantly reduces the search space and computational complexity compared to exhaustive search methods. Despite its advantages, LSH has certain limitations. The choice of hash functions is critical; inappropriate selection can lead to suboptimal performance, with similar items being hashed into different buckets. Moreover, LSH can require substantial memory resources, especially when dealing with multiple hash tables and large datasets. To mitigate these challenges, alternative methods such as data-dependent hashing techniques, including spectral hashing and deep hashing, have been developed. These approaches aim

to learn optimal hash functions from the data, potentially offering improved accuracy and efficiency in specific applications.

Tree-based methods, such as SPTAG [12] and ANNOY [2], use hierarchical data structures to partition the data space into nested regions, facilitating efficient search. SPANN (Scalable and Practical Approximate Nearest Neighbour) [13] stores centroid points of clusters in memory while keeping the extensive posting lists on disk. As shown in Figure 2.2, during index construction, SPANN employs a hierarchically balanced clustering algorithm to equalize posting list lengths and enhances them by including points from the closure of corresponding clusters. For querying, it dynamically prunes unnecessary posting lists based on the query, optimizing search performance. ANNOY (Approximate Nearest neighbours Oh Yeah) constructs multiple binary trees (a forest) where each tree is built by recursively splitting the data using random hyperplanes. During a query, a subset of these trees is traversed to gather candidate neighbours, which are then ranked to find the approximate nearest neighbours. A significant challenge for tree-based methods is their performance degradation in high-dimensional spaces, due to the "curse of dimensionality." As the number of dimensions increases, the data becomes more sparse, and the distance metrics lose their effectiveness, leading to a decline in the efficiency of these tree structures. In such scenarios, the partitioning strategies employed by these methods may become less effective, resulting in increased search times and reduced accuracy. Another limitation is the sensitivity of tree-based methods to data distribution. They often assume a balanced distribution of data for optimal performance. However, in cases where the data is skewed or clustered unevenly, these methods can become inefficient. The tree structures may become unbalanced, leading to longer search paths and increased computational overhead. Additionally, the process of balancing these trees can be complex and time-consuming, further impacting performance.

Quantization-based methods such as product quantization (PQ) [37] compress the high dimensional vectors into short codes to reduce the space overhead in ANN search. As shown in Figure 2.3, the process begins by dividing each input vector into M equal-length subvectors. For each subvector, a separate codebook is created by applying a clustering algorithm, such as k-means, which identifies K cluster centroids representing the subspace. During encoding, each subvector is replaced by the index of its nearest centroid from the corresponding codebook. This results in a compact representation of the original vector as a sequence of centroid indices. To reconstruct an approximate version of the original vector, the stored indices are used to retrieve the corresponding centroids from each codebook, which are then concatenated. By quantizing each subvector independently, PQ effectively reduces the computational complexity and memory footprint associated with storing and searching high-dimensional data.

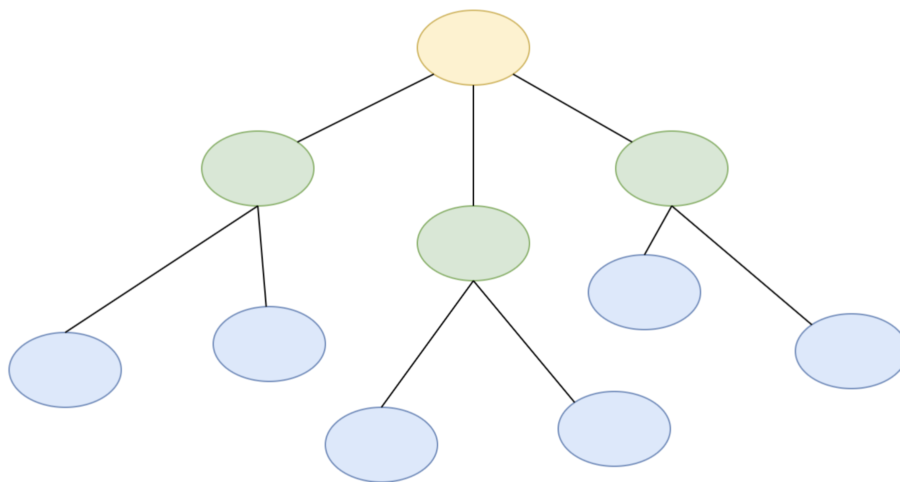


Figure 2.2: Hierarchical balanced clustering of SPANN [13]. The vectors in the large (yellow) cluster are iteratively balanced and partitioned into a few smaller (green) clusters, continuing this process until each resulting cluster contains only a limited number of vectors (blue clusters).

The primary concern is the introduction of quantization errors during the compression process. By representing high-dimensional vectors with indices of their nearest centroids, PQ inherently approximates the original data, which can lead to decreased recall and accuracy in search results. This approximation may be unsuitable for applications where high precision is critical. Furthermore, training codebooks requires representative static datasets, making dynamic updates inefficient as adding new data often necessitates retraining, and its performance is highly sensitive to parameter choices like the number of centroids; too few erode precision, while too many bloat computational costs during encoding and querying. Though PQ reduces memory usage, query efficiency relies on asymmetric distance approximations that trade speed for accuracy, and its rigid subspace partitioning struggles with heterogeneous data distributions, unlike adaptive methods.

2.3 Graph-based Indexes and ANNS

A graph-based index builds a directed graph $G = (V, E)$ as the index, where each vertex $v_i \in V$ represents a vector in the dataset and each directed edge $e_j \in E$ represents the

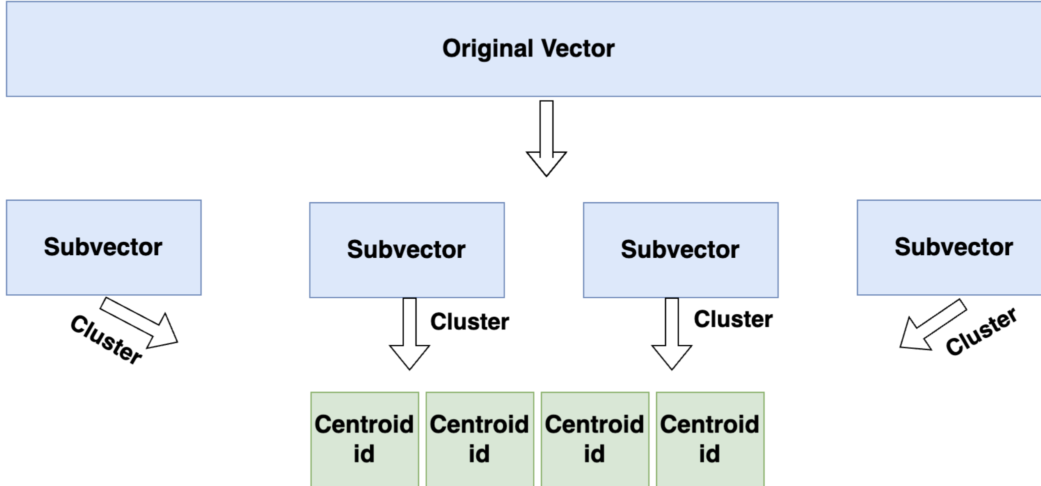


Figure 2.3: Product Quantization

proximity of two vectors. Let (u, v) be a pair of vertices in the graph corresponding to two vectors and let $\delta(u, v)$ be the distance between the vectors. The edge connecting u and v is considered short or long based on the value of $\delta(u, v)$ [25]. This is called the *proximity graph*. Figure 2.5 illustrates a simple example of a graph-based index.

ANNS using graph-based indexes typically performs a greedy beam search [64] over the proximity graph in order to find the nearest neighbours of a query (Algorithm 1). Depending on the method, the search begins either at a random or chosen entry point and iteratively moves closer to the target. During this procedure, two priority queues are used, C and K, which are sorted according to proximity to the query. The queue C holds the points scheduled for examination, while K keeps only the closest k points that have already been evaluated. At each iteration, the algorithm removes the foremost point from C to inspect all its neighbours. Should the distance from any of these neighbouring points to the query be less than that of the farthest point in K, that neighbour is added to both C and K. The process terminates once the nearest point in C is further away than any point in K. Ultimately, the algorithm outputs the elements found in K.

A skewed dataset distribution leads to the presence of hubs (vertices with high out-degrees) within the graph. These hubs have a high visit frequency and are more likely to link to other hubs [64], which can impact the navigability of the graph, increasing the likelihood

Algorithm 1 Search on Proximity Graph [68]

Require: A proximity graph $G = (V, E)$, a query point q, and the number of k nearest neighbours to return
Ensure: A set K of k nearest neighbours of q in V

```
1:  C := ∅;                                   ▷ Candidates sorted closest to furthest from the query
2:  K := ∅;                                   ▷ top k result so far sorted furthest to closest from q
3:  V := ∅;                                   ▷ Hashmap storing the visited points
4:  for a start vertex v_s;
5:  C := C ∪ {v_s}; V := V ∪ {v_s};
6:  while |C| > 0 do
7:      v_c := C.Min(); C.Pop();                                  ▷ pop the point v_c
8:      v_f := K.Max();                                           ▷ k-th closest neighbour so far
9:      if δ(v_c, q) > δ(v_f, q) and K.size() = k then            ▷ Ending condition
10:         break;
11:     end if
12:     if K.size() > k then
13:         K.Pop();
14:     end if
15:     K := K ∪ {v_c};
16:     for each outgoing neighbour u of v_c in G do
17:         if u ∉ H then
18:             C := C ∪ {u}; V := V ∪ {u};
19:         end if
20:     end for
21: end while
22: return K;
```

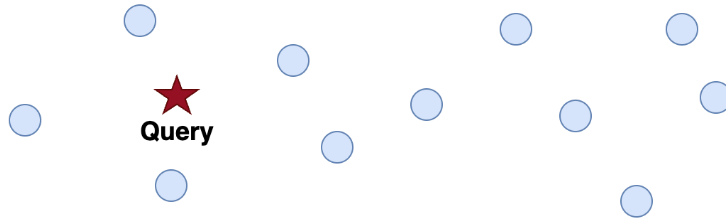


Figure 2.4: Original Dataset

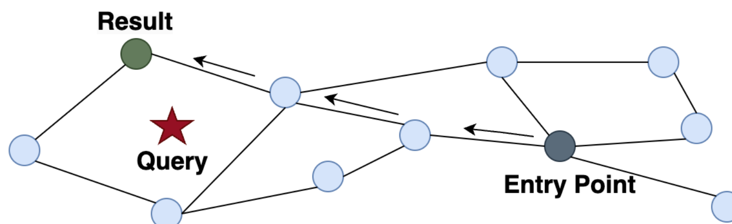


Figure 2.5: Graph-based index performing ANNS to find the closest neighbour to the query

of the greedy search to get caught in a local optimum. To ensure effective navigability, hubs require a high degree and a greater number of long-range connections. A graph-based index should ideally contain a mix of long edges, which allow speedy navigation from the starting point towards the region close to the query point, and short edges, which allow the search to converge quickly in the target region and improve accuracy. One way of doing this is to prune certain edges. The RNG Strategy [58] is a method for selecting edges of a graph for ANNS. Given a pair of vertices $(u, v) \in P$ and the distance function $\delta(u, v)$, the RNG is a graph constructed as follows: an edge from u to v is included in the RNG if and only if $\forall u' \in P$, u' cannot prune the edge (u, v). In particular, u' can prune (u, v) if and only if u' is closer to u than v and also closer to v than u, i.e., $\delta(u, u') < \delta(u, v)$ and $\delta(v, u') < \delta(u, v)$. In other words, as shown in Figure 2.6, the RNG rule removes the longest edge in a triangle in the graph.

In the next sections we will discuss different strategies that graph-based indexes use to construct a graph appropriate for ANNS.

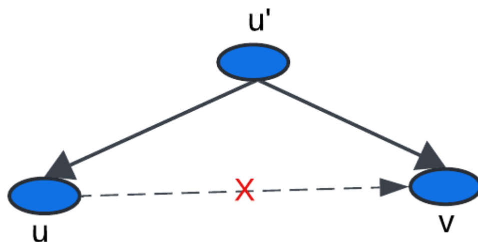


Figure 2.6: RNG rule

2.4 Refinement-based Graph Construction

Refinement-based graph construction starts with an initial graph and then undergoes a refinement process to improve the quality of the graph (Figure 2.7). The initial graph is constructed with each vertex randomly connected to k neighbours. Index construction is fast primarily because the randomly-connected graph is available at the start and is refined iteratively to improve its quality, rather than building the graph incrementally. Instead of performing global searches for connections, these methods refine the graph iteratively by focusing on the immediate neighbourhoods of vertices [63, 40]. Consequently, the number of distance computations while building the index are moderated.

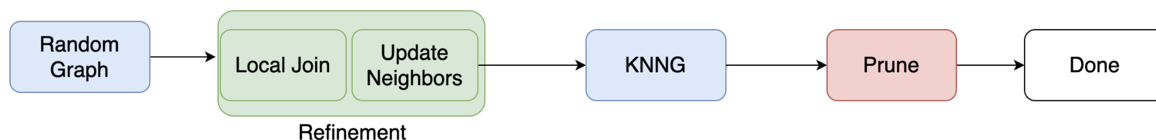


Figure 2.7: Refinement-based construction

The simplest refinement-based graph structure is K-Graph [24], where each vertex out-links its k nearest neighbours in the graph. K-graph suffers from poor query performance due to a loss of navigability caused by too high of an out-link density [64], leading to a search getting caught in a local optima/hubs before reaching the global optima. This issue is a common theme in graph-based indexes. While local connections are important for the final recall stage of a search, too many local connections lead to a higher chance of getting stuck in local optima and lowering search accuracy.

Since constructing an exact K-Graph is unrealistically time consuming, most solutions turn to NN-Descent [24] to construct an approximate KNN graph. NN-Descent is an iterative, heuristic method for building an approximate $k-$nearest neighbour graph (kNNG). While a brute-force method has a computational cost of $O(n^2)$ where n is the number of vertices in the dataset, NN-Descent typically operates with a lower empirical cost of $O(n^{1.14})$ [11]. The main idea is the assumption that "the neighbour of my neighbour is likely to be my neighbour." If vertices v_0 and v_1 are not yet neighbours but are both in the nearest neighbour list of a third vertex v_2, then v_0 and v_1 are likely to be near each other. As a result, v_2 suggests that v_0 and v_1 compare their distances and update their nearest neighbour lists if necessary. This process is shown in Figure 2.8. NN-Descent begins by randomly initializing a $k-$NNG and continues to check neighbours until the number of newly identified close neighbours drops below a specified threshold. As shown in Figure 2.8, in the sampling phase only the partial neighbours of vertex 1 are sampled in order to eliminate duplicate computations. The computation phase starts after all vertices have been sampled. The similarity value between vertices 2 and 5 is computed by loading the vectors of each of them, computing each dimension in turn, and summing up to get the distance value Dist1. The computation process for vertex pairs (2,7) and (5,7) is the same. After the similarity value between any vertex pair is computed, the neighbour list of both vertices will be updated. For vertex 2, vertex 5 is a candidate neighbour. The algorithm determines whether to insert vertex 5 into the neighbour list of vertex 2 by comparing Dist1 with the existing similarity values of vertex 2. Vertex 2 is also treated as a candidate vertex to update the neighbour list of 5. Similarly, after the vertex pairs (2,7) and (5,7) are computed, the neighbour lists of vertex 2, vertex 5, and vertex 7, respectively, will be updated.

There are many algorithms that use NN-Descent in their approaches to build an approximate KNN graph and then use edge-selection strategies to improve performance. One popular such index is NSG [18], which first builds a graph using NN-Descent and then prunes using an edge selection strategy based on the RNG rule. Other examples include Efanna [17], which first builds multiple KD-trees [51] and then executes NN-Descent, and Cagra [41], which builds a graph using NN-Descent on a GPU and then prunes edges based on an edge-selection strategy.

A recent construction strategy called Relative NN-Descent (RNNDescent) [40] has been shown to construct graphs much faster than existing methods, however it can have lower search performance [49]. RNNDescent combines NN-Descent and the RNG rule to speed up the graph construction. It prunes during the initial graph construction process, as opposed to other strategies that postpone pruning until after the creation of the KNN graph. This allows for the graph to be constructed much faster. All the solutions mentioned above add

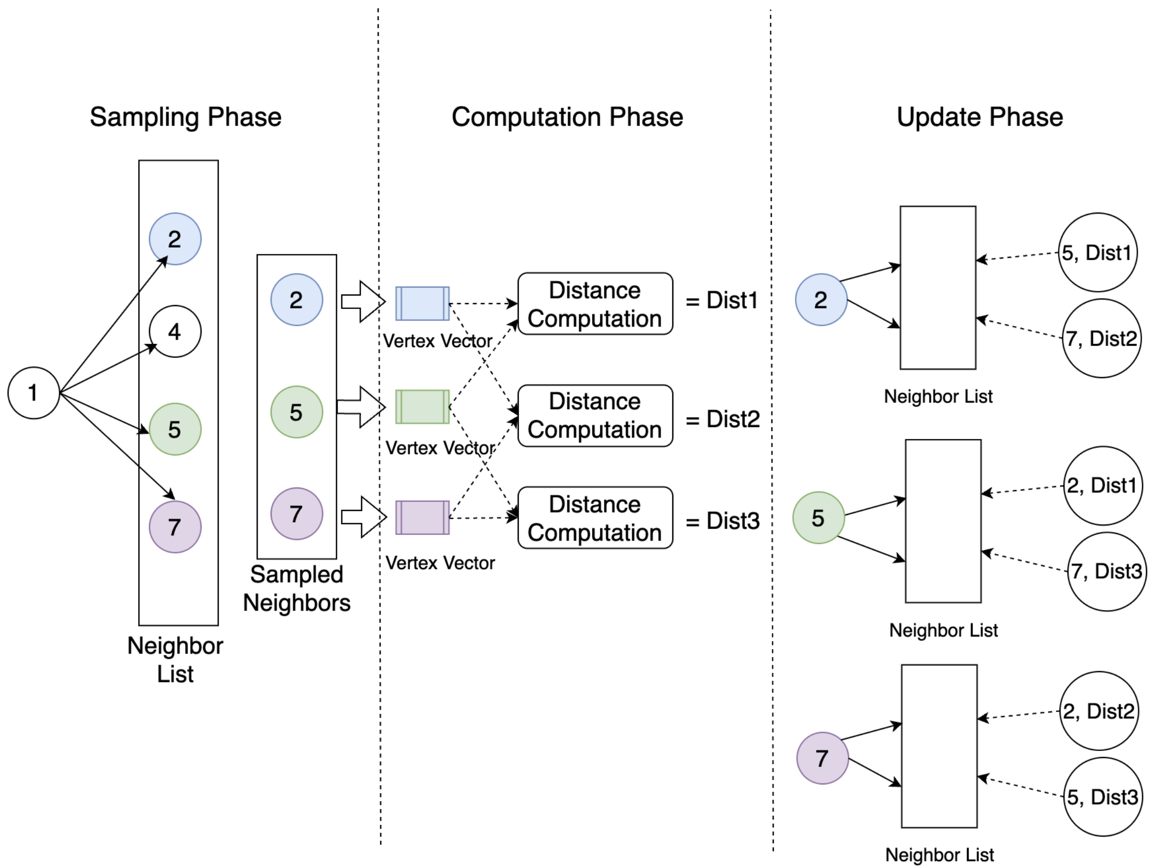


Figure 2.8: Simple Illustration of NN-Descent join.

bi-directional/reverse edges to the graph in order to enhance the graph navigability and reduce the number of strong connected components.

2.5 Increment-based Graph Construction

Increment-based graph construction traditionally starts with an empty graph and treats every point being added as a query, performing ANNS on the current graph to find the k nearest neighbours of the newly inserted point (Figure 2.9).

The earliest increment-based construction algorithm is NSW (Navigable Small Worlds)

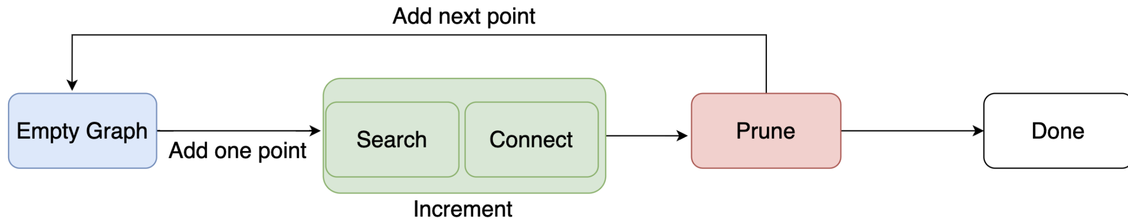


Figure 2.9: Increment-based construction

[35], which introduces long-range connections to break up hubs, enhancing the connectivity of the graph. The way NSW forms these connections is with randomization. When inserting a vertex (i.e., a vector in P), NSW performs the greedy search (Algorithm 1) to find neighbours and adds reverse edges from those neighbours. Longer edges will most likely be constructed in the early phases of construction and later become links between the hubs that keep the overall graph connected and allow the logarithmic scaling of the number of hops during greedy routing. Search over NSW can still get caught in a local optima, because the maximum out-degree for each vertex of the graph is not fixed.

HNSW (Hierarchical Navigable Small Worlds) [34], addresses this issue by building a hierarchical structure: building NSW graphs at multiple layers. As shown in Figure 2.10, the bottom layer stores all the vertices in the graph and the number of vertices decreases as higher layers are constructed. The number of points in each layer is reduced exponentially as one moves up, keeping the top layers for long edges and speed, and the bottom layers for shorter edges and accuracy. The idea is based on the skip-list [48], a hierarchical sorted linked list structure that allows for searching and inserting in $O(\log n)$ time complexity on average.

Vertices in HNSW are inserted one-by-one. Every vertex is assigned an integer l, following an exponentially decaying probability distribution that indicates the maximum layer in which the vertex can be located. For example if $l = 2$, the vertex can be in layers 2, 1 and 0. Once a vertex is assigned an l value, the algorithm starts from the highest assigned layer and performs a nearest neighbour search using the same greedy search strategy mentioned above, looking at C candidates, which is a parameter set by the user, and represents the size of the candidate set C from Algorithm 1. Once C neighbours are found, M (a user-set variable) neighbours are connected to the vertex and the algorithm moves to the next lower layer and continues the process. The process ends when the same search is conducted at the bottom layer (Layer 0). To disrupt the skewness and hubs

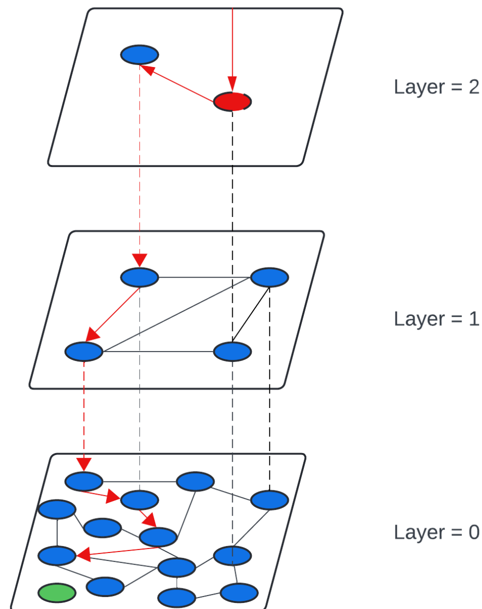


Figure 2.10: Illustration of HNSW. The search path is shown by red arrows with a fixed entry points and the query point is in green.

mentioned earlier, HNSW follows NSW and uses randomization, and also introduces an out-degree limit and RNG pruning to break up hubs.

During search, HNSW starts from the highest layer and performs the greedy beam search until the local nearest neighbour to the query point is found (say w at that layer). Once that happens, the algorithm moves down to the next layer and repeats the same process starting from the vertex that corresponds to w. This is repeated down to the lowest layer where k nearest neighbours are found.

HNSW is the most popular graph-based ANNS algorithm due to its superior search performance, both in query efficiency and accuracy, on hard datasets (i.e., datasets whose local intrinsic dimensionality (LID) is high) [63]. It also allows for incremental inserts, as a new point can be added at any time in the algorithm, without requiring any sort of rebuilding.

Increment-based approaches provide high graph-quality and search performance due to enhanced connectivity [63], while also allowing for incremental inserts, however they have long index build times. Refinement-based approaches are quicker to build, but require

a full graph reconstruction for any new inserts and can suffer in search quality on hard datasets if they do not perform any post-processing after refinement. In the next section we introduce MIRAGE, which combines the best of both approaches to construct indexes fast and deliver high quality searches.

2.6 Other Related Works

We discussed background related to vector indexing and ANNS in Sections 2.2 and 2.3. In this section we focus on recent works that focus on improving the performance of these methods.

Vamana [52] is the graph-based algorithm that serves as the foundation for the DiskANN system. The construction of a Vamana graph begins with an initialization phase where each vertex is assigned a set of random out-neighbours. Then, the algorithm iterates through all points in a random order, running a greedy search to identify candidate neighbours and applying a robust pruning mechanism to refine the edges. A key feature of Vamana is its tunable α parameter, which controls the trade-off between graph connectivity and search efficiency by enforcing that each step in a search path results in a multiplicative distance reduction. The algorithm makes two passes over the data, first with $\alpha = 1$ and then with a user-defined α greater than 1, ensuring that long-range connections are established for improved search performance. The search process in Vamana follows a best-first traversal strategy where it starts at a designated entry point (usually the dataset medoid) and greedily explores nodes, guided by distance comparisons. The key advantage of Vamana lies in its ability to construct graphs with fewer search hops compared to other methods, making it well-suited for both in-memory and SSD-based indices.

The Hierarchical Voronoi Structure (HVS) [33] is a graph that begins by partitioning the space into hierarchical Voronoi cells, where each data point belongs to exactly one cell. This results in a hierarchy of Voronoi diagrams, with seed points from higher layers appearing in lower layers. In each layer, a graph is constructed where nodes correspond to Voronoi cells, and edges are formed based on proximity between cells. The edge selection strategy follows principles similar to HNSW. To improve efficiency, a density-based allocation strategy is used, ensuring that fewer data points are included in deeper layers. The search process in HVS follows a top-down hierarchical traversal approach combined with a base-layer graph search. First, a multi-level distance book is computed to store distances between the query and Voronoi cell representatives at different levels. The search begins at the coarsest level, identifying the nearest Voronoi cell as the global entry point. It then progresses layer by layer, where each visited cell's representative determines the entry point for the next lower

layer. If a locally nearest Voronoi cell is found before reaching the base layer, the search can terminate early, optimizing efficiency.

LSH-APG (Locality-Sensitive Hashing Approximate Proximity Graph) [70] is an approach that combines locality-sensitive hashing (LSH) with graph-based search methods. LSH-APG constructs a proximity graph using a increment-based approach. Instead of relying solely on traditional graph-based methods, LSH-APG uses LSH to preselect entry points for each new insertion. Additionally, an LSH-based pruning condition is used to filter out distant neighbours during insertion, further reducing unnecessary computations. The search process in LSH-APG starts by using the LSH index to quickly retrieve high-quality entry points in the graph. Once a good starting point is identified, a greedy graph traversal is performed, where the query moves iteratively towards closer neighbours in the APG.

The SISAP Indexing challenge [54, 55] is a competition where participants submit indexing methods of all types, and compete on construction time and search performance. One method that stands out in these competitions is The Continuous Refining Exploration Graph (crEG) [25], which introduces a dynamic pruning strategy based on the average neighbour distance, continuously refining edge quality during and after construction. Similarly, a new framework [67] for proximity graph construction introduces the alpha-pruning strategy, which incorporates angular constraints to improve neighbour selection and graph sparsity. Both methods aim to improve upon pruning using only the RNG rule. MIRAGE uses the RNG rule for pruning, making it easier to be adapted into existing libraries, but could also be adjusted to different methods of pruning.

In addition to algorithmic advancements, there are solutions that aim to improve graph-based ANNS performance by using hardware accelerators. SONG [69] is a GPU-based ANNS implementation that offers a significant speedup over CPU counterparts, outperforming FAISS by leveraging GPU-centric strategies. GANNS [68] builds on this by enhancing parallelism and using an NSW-based proximity graph. More recently, CAGRA [41] has utilized modern GPU techniques, such as warp splitting, to achieve substantial performance improvements over other GPU-based methods. However, these graph-based approaches, including SONG, GANNS, and CAGRA, face limitations when scaling to larger datasets, as they require the entire graph index to reside in GPU memory.

Beyond GPUs, there has been use of other accelerators like FPGAs or ASICs for ANNS. FPGA-based methods have been investigated, e.g., vStore [30] and TPU-KNN [14].

Chapter 3

MIRAGE Index

In this chapter we describe MIRAGE, a new graph-based index that provides the fast construction of refinement-based approaches and the superior search performance and incremental insert capabilities of increment-based approaches.

3.1 Motivation and Intuition

The intuition behind MIRAGE lies in understanding the keys to a high-performing graph-based index: efficient construction, global navigability, local precision. As mentioned in Section 2.3, graphs work well for ANNS when they allow a search to quickly traverse distant regions of the graph to locate the general vicinity of a query point, and then refine the search with precision within that local neighbourhood. However, this balance needs to be achieved while maintaining fast index construction.

In order to ensure efficient construction, MIRAGE indexes all points starting with a randomly connected graph and progressively refines it while pruning at the same time (Figure 3.1). It follows the refinement-based approach which allows for fast construction times due to the reasons mentioned in Section 2.4.

The challenge for solutions following refinement is that the search over the graph could potentially get caught in local optima reducing their search performance (quality and queries-per-second) unless they perform some post-processing, such as NSG. MIRAGE addresses global navigability and local precision by incrementally building upper layers with a subset of points that contain long-range links that enable rapid traversal across the graph. As the hierarchy levels increase, the graphs get sparser. This hierarchical structure

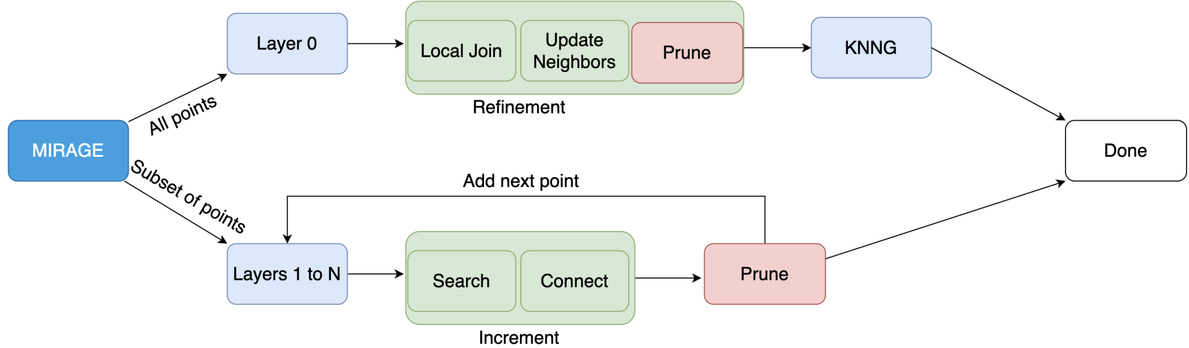


Figure 3.1: MIRAGE Construction

allows searches to begin in the upper layers for efficient broad exploration, and progressively narrow down to precise results in the bottom layer, ensuring good search performance.

Consequently, MIRAGE builds a hierarchical structure, where the bottom layer (Layer 0) of the graph is constructed using a refinement-based approach, and then creates layers on top (Layers 1 to N) following an increment-based approach. During search, MIRAGE uses the greedy beam search, starting from the top layer and moving down until the bottom layer is reached where the k nearest neighbours to the query point are found. New inserts to the constructed structure occur in the same manner as increment-based construction, going through ANNS to find their correct neighbours.

3.2 Graph Construction

The construction of MIRAGE is given in Algorithm 2 and follows two steps: Layer 0 construction (lines 1-10) and the construction of Layers 1 to N (lines 12-28).

It constructs the bottom layer (Layer 0) of the graph using a refinement-based approach. To be exact, it initializes a graph connected to S random neighbours and then runs Algorithm 3, which finds neighbours using a traditional NN-Descent algorithm and prunes edges at the same time using the RNG rule. As opposed to a traditional refinement-based approach that automatically forms edges between neighbours of neighbours (i.e., two-hop neighbour edges), we use insights from RNNDescent to first check whether the RNG rule is satisfied before adding the edge (lines 11-13 of Algorithm 3). This speeds up construction time by eliminating unnecessary addition of edges that would be pruned later anyway. We

Algorithm 2 MIRAGE Construction

Require: Dataset P, S, M, C, $Iter$, R level normalization $m_L \in (0, 1)$
Ensure: Hierarchical graph structure $G = (L_0, L_1, \ldots, L_\ell)$
1: Initialize Layer 0: $L_0 \leftarrow$ RandomGraph(S)
2: Set all flags in L_0 to "new"
3: **for** $r \leftarrow 1$ to R **do**
4:     **for** $iter \leftarrow 1$ to $iter$ **do**
5:         Update(L_0, M)
6:     **end for**
7:     **if** $r \neq R$ **then**
8:         AddReverseEdges(L_0)
9:     **end if**
10: **end for**
11: $L_0 \leftarrow$ Constructed bottom layer
12: $entry_point \leftarrow$ Choose a random node from L_0
13: **for** each point $p \in D$ **do**
14:     $level \leftarrow$ RandomLevel(m)
15:     **if** $level > 0$ **then**
16:         **for** $l \leftarrow$level to 1 **do**
17:             $closest \leftarrow$ Algorithm 1$(p, entry_point, 1, l)$
18:             $entry_point \leftarrow$ nearest element from $closest$ to p
19:         **end for**
20:         **for** $l \leftarrow 0$ to $level$ **do**
21:             $neighbours \leftarrow$ Algorithm 1$(p, entry_point, C, l)$
22:             $selected_neighbours \leftarrow$ RNG Rule(layer l, M)
23:             AddReverseEdges(layer l)
24:         **end for**
25:     **end if**
26:     **if** $level >$ max level in G **then**
27:         Update entry point: $entry_point \leftarrow p$
28:     **end if**
29: **end for**
30: **return** G

maintain a flag for each vertex indicating whether it is "new" or "old" to help eliminate duplicate distance comparisons of the same vertices – a common practice in refinement-based approaches. MIRAGE then follows the standard practice in graph-based indexes and adds reverse edges to complete the bottom layer of the graph (line 8 of Algorithm 2). We limit the maximum out-degree of the constructed graph, pruning every vertex in Layer 0 down to M neighbours using the RNG rule (lines 21 and 22 of Algorithm 3), something that RNNDescent does not do. This helps improve MIRAGE's reachability and search performance. Due to its fast construction time, use of the RNG rule to select edges, and its use of reverse edge addition, the refinement-based constructed graph at Layer 0 is best suited to be incorporated into an increment-based approach for the higher layers.

To generate upper layers, a maximum layer index L is chosen using an exponentially decaying probability distribution, normalized by $m_L = \frac{1}{\ln(M)}$, as is done in HNSW. Each point in the dataset is assigned a layer by the distribution and then first inserted at the corresponding layer, where it finds its M nearest neighbours using Algorithm 1, moves down a layer and repeats the same process, stopping after being inserted at layer 1. While selecting neighbours at each layer, if the value of M is exceeded for any vertex, we use the RNG rule to prune edges. We also add reverse edges to the graph to improve navigability. Using ANNS in the construction of the upper layers allows us to get the best of both worlds in terms of refinement and increment-based construction. Since the bottom layer is constructed using refinement-based approach over the entire dataset, and the construction of hierarchy uses ANNS on only a subset of the dataset, we get the improved speed of the refinement-based approach, with the improved connectivity of using an increment-based approach for the upper layers. This ensures that, during search, we get optimal entry points at each layer, leading to improved query accuracy and efficiency.

3.3 Search

MIRAGE's search algorithm follows most graph-based indexes, namely standard greedy search, with the addition of hierarchy. It starts with a pre-defined entry point at the top-most layer of the graph and then performs Algorithm 1 on each layer with $k = 1$, i.e., until it finds a single nearest neighbour to the query on said layer. Then it moves to the same point on the layer below and repeats the process, finally concluding at the lowest layer once the k nearest neighbours to the query are found. Intuitively this process allows for the best entry point at each layer, helping improve query accuracy and throughput. During search, the value of C is of the utmost importance. Recall that C represents the size of the candidate list, i.e. how many neighbours to explore during the search. A higher

Algorithm 3 UPDATE

Require: Graph $G = (V, E)$, vertex $u \in V$, maximum out-degree M
Ensure: Updated edge set $E \subset V \times V$

```
1:  for each u ∈ V do
2:      U ← {v | (u, v) ∈ E}
3:      Sort v ∈ U in ascending order of δ(u, v)
4:      U′ ← ∅
5:      for each v ∈ U do
6:          f ← true
7:          for each w ∈ U′ do
8:              if both flags of v and w are "old" then
9:                  continue
10:             end if
11:             if δ(u, v) ≥ δ(v, w) then // RNG rule check
12:                 f ← false
13:                 E ← (E \ (u, v)) ∪ (w, v)
14:                 break
15:             end if
16:         end for
17:         if f then
18:             U′ ← U′ ∪ {v}
19:         end if
20:     end for
21:     if |U′| > M then
22:         RNG RULE(U′, M)
23:     end if
24:     Set the flag "old" for all vertices in U′
25: end for
```

C will result in a slower but more accurate search, whereas a lower C will result in a faster but less accurate search.

Algorithm 4 MIRAGE Search

Require: Multi-layer graph G, query element q, number of nearest neighbours to return K, size of the dynamic candidate list C
Ensure: K nearest elements to q
1: $N \leftarrow \emptyset$ ▷ Set for the current nearest elements
2: $ep \leftarrow$ get entry point for MIRAGE
3: $L \leftarrow$ level of ep ▷ Top layer for MIRAGE
4: **for** $lc = L$ **to** 1 **do**
5: $W \leftarrow$ Algorithm 1$(q, ep, C = 1, lc)$
6: $ep \leftarrow$ get nearest element from W to q
7: **end for**
8: $W \leftarrow$ Algorithm 1$(q, ep, C, lc = 0)$
9: **return** K nearest elements from N to q

3.4 Handling Updates

Vector datasets can often evolve over time, requiring indexing approaches that can efficiently support dynamic updates [50]. These updates include insertions and deletions, though in-memory graph-based methods typically overlook deletions [33]. Consequently, in this thesis we focus on handling incremental inserts in MIRAGE and leave incremental deletions for future work. As noted earlier, refinement-based solutions such as NN-Descent, RNNDescent, and NSG do not handle incremental insertions.

For incremental inserts, MIRAGE follows the same method it uses for Layers 1 to N and starts by assigning a layer to the new point. Next it inserts that point at its highest layer, performs Algorithm 1 to find its M nearest neighbours, and moves down a layer to repeat the same process. The difference from construction (Algorithm 2) is that an update does not stop at Layer 1, but proceeds to Layer 0. This is because we want all the points to be present at Layer 0, since that is where the search concludes. If any point exceeds M connections, RNG rule is used to prune edges. Using an increment-based approach for incremental inserts allows us to preserve the quality of the graph and ensures that we do not have to reconstruct the graph from scratch. To effectively compare the accuracy of insertion-based solutions such as MIRAGE and HNSW, existing literature states that it

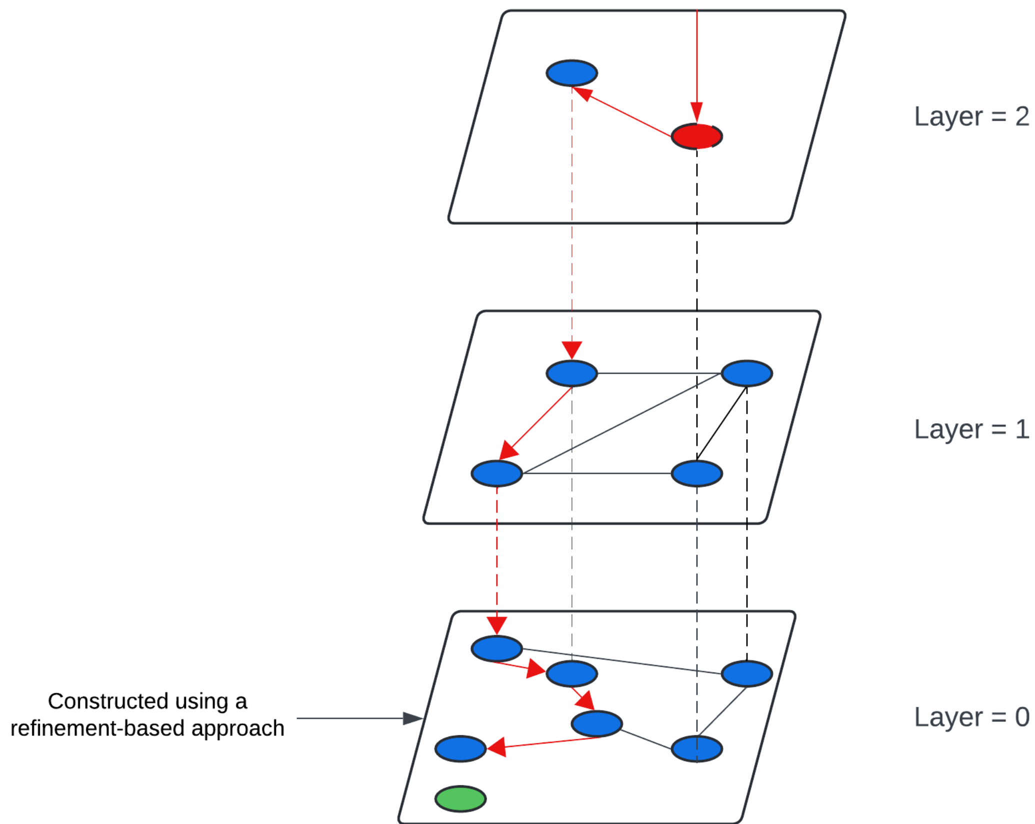


Figure 3.2: Diagram of the MIRAGE index. The search path is shown by red arrows with a fixed entry points and the query point is in green. Due to Layer 0 being constructed with a refinement-based approach, MIRAGE is much faster than HNSW in terms of indexing speed. MIRAGE also has a sparser bottom layer which allows for it's search efficiency to be better than HNSW, as much as 2x.

is sufficient to evaluate their KNN search performances, as the neighbours of each newly inserted point are selected from the approximate nearest neighbours returned by the search [33]. As shown in our experimental results, MIRAGE delivers faster query results with the same precision as HNSW, indicating that MIRAGE can handle insertions efficiently.

3.5 Complexity of Algorithms

The overall construction complexity of MIRAGE is: $O(n^{1.14})$, where n is the number of vertices in the dataset.

The construction of the bottom layer dominates the overall construction time with an empirical complexity of approximately $O(n^{1.14})$. After Layer 0 is constructed, the increment-based algorithm is used to build additional layers hierarchically. The complexity of this construction alone is $O(n \cdot \log(n))$, but since the construction of the bottom layer has a higher complexity, the total construction complexity remains $O(n^{1.14})$.

The search complexity is primarily influenced by the hierarchical nature of the algorithm and is $O(\log(n))$. The search process proceeds layer-by-layer, utilizing a greedy search strategy to progressively narrow down the search space until it reaches the bottom layer. The initial guidance provided by the upper layers significantly localizes the search area, ensuring that the total search time scales logarithmically with the size of the dataset.

We note that neither HNSW, NN-Descent, nor MIRAGE provide theoretical guarantees on graph connectivity; therefore, we rely on empirical results in the next chapter for the analysis of our method's effectiveness [44].

Chapter 4

Evaluation

We evaluate MIRAGE through a series of experiments on real world datasets.

4.1 Experimental Setup

Our experiments involve comparing MIRAGE with existing methods in terms of construction and search performance, looking at the comparative qualities of the graphs created by different methods, and verifying the effectiveness of our method with ablation studies.

Datasets. We conduct our experiments on 7 real world datasets with different sizes and dimensions (See Table 4.1). These datasets are commonly used for the evaluation of different ANNS indexing methods. Gist1M [4] consists of 1 million high-dimensional (960-dimensions) GIST descriptors extracted from images, along with 1,000 query vectors. The SIFT10M dataset [4] consists of a collection of 10M and 100M base vector respectively, and 10K query vectors. All of the vectors are 128-dimensional local SIFT descriptors from INRIA Holidays images. The Paper dataset [61] consists of about 2M base vectors and 10K query vectors. The dataset is generated by extracting and embedding the textual content from an in-house corpus of academic papers. Crawl dataset [3] provides a corpus for collaborative research, analysis and education. The text is converted to 300-dimension vectors with fastText55. Glove (Global Vectors for Word Representation) [47] is a word vector dataset produced by the corresponding Glove unsupervised learning algorithm. This dataset is the pre-trained word vectors from Wikipedia 2014. Tiny image [1] dataset extracts 5 million images from 80 million images of size 32×32 pixels collected from the Internet, crawling the words in WordNet. Deep10M consists of the first million vectors

of the Deep1B dataset [66]. The vectors were obtained by running a convolutional neural network on an image collection, reduced to 96 dimensions by principal component analysis and subsequently l2-normalized.

Dataset	**Dimension**	**# Base**	**# Query**	**LID**	**Type**
Tiny5M	384	5,000,000	1,000	-	Image
Deep10M	96	10,000,000	10,000	12.1	Image
SIFT10M	128	10,000,000	10,000	9.3	Image
GIST1M	960	1,000,000	1,000	18.9	Image
Crawl	300	1,989,995	10,000	15.7	Text
GloVe	100	1,183,514	10,000	20.0	Text
Paper	200	2,029,997	10,000	-	Text

Table 4.1: Summary of datasets with dimensions, base size, query size, LID, and data type. LID [29] denotes how many "degrees of freedom" are necessary to describe the dataset and is used to measure the difficulty of answering NN queries over it; higher LID means harder dataset.

Performance Metrics. To measure each method's overall performance, we employ metrics related to index construction and search. For index construction, we evaluate the index construction time and peak memory usage. Some index characteristics, such as average out-degree and the number of connected components, are recorded; they indirectly affect index construction efficiency and size. For search, we evaluate search speed and accuracy. Search speed is measured in queries per second (QPS). QPS is the ratio of the number of queries ($\#q$) to the search time (t) : $\frac{\#q}{t}$. We evaluate search accuracy using two metrics: (1) recall, which measures the ratio of successfully retrieved ground truth k-nearest neighbours to k (see Definition 4 above) and (2) average distance ratio (ADR) (See Definition 5 above) [19, 26, 45, 53, 46, 20], which represents the mean of the distance ratios (equivalent to the average relative error on distance plus one) of the retrieved k objects relative to the ground truth k-nearest neighbours.

Methods and Parameters. We select 6 graph-based indexing methods to compare against MIRAGE, namely HNSW [34], NN-Descent [24], NSG [18], RNNDescent [40], Vamana (DiskANN) [52], and LSH-APG [70]. We selected HNSW and NSG because of their superior search performance in multiple studies and their popularity. HNSW is also the most popular increment-based graph index. NN-Descent is the most popular refinement-based approach and is used as the first step in many graph-based indexes including NSG. The other three: RNNDescent, Vamana, and LSH-APG, are recent proposals optimized for index construction time.

We configure each method's parameters to maximize its search performance. For the Sift, Gist, Glove, and Crawl datasets we use the parameters provided by [63], which are found by using a grid search within the parameter space. For the Deep dataset we follow [52] where the parameters are found using a parameter sweep. For the Tiny dataset, we follow guidance from [33], and for the Paper dataset we follow the parameters set by the original paper [61].

Platform. Experiments are run on a Linux server of Xeon(R) Platinum 8380 CPU and 1 Terabyte of DDR4 RAM. In our experiments we set the thread count to 128. We implement the MIRAGE index on the FAISS [16] library in C++. For the HNSW, NN-Descent, RNNDescent, and NSG indexes we benchmark against the existing Faiss implementations in order to make sure the underlying system is the same. For the Vamana and LSH-APG, we benchmark using their existing C++ implementations.

Our code is available at: https://github.com/dsg-uwaterloo/mirage

4.2 Memory Cost

Figure 4.1 shows the peak memory needed to construct each index, which is different from index size. We choose this metric because a previous study [33] indicates that index size mainly depends on the average out-degree of the graph, and the smaller the value, the smaller the index size [64]. For completeness we show those values and discuss them in Section 4.5. Because of this, methods that prune edges using methods such as the RNG rule have fewer edges and therefore lower index size. However, this does not tell the whole story – in order to generate graphs using a refinement-based approach, many indexes start with the K-Graph approach, which has a high maximum out-degree during the construction process, incurring high memory cost. This applies to approaches like NSG and NN-Descent [33]. Therefore, following the current literature, we consider memory cost in addition to index size.

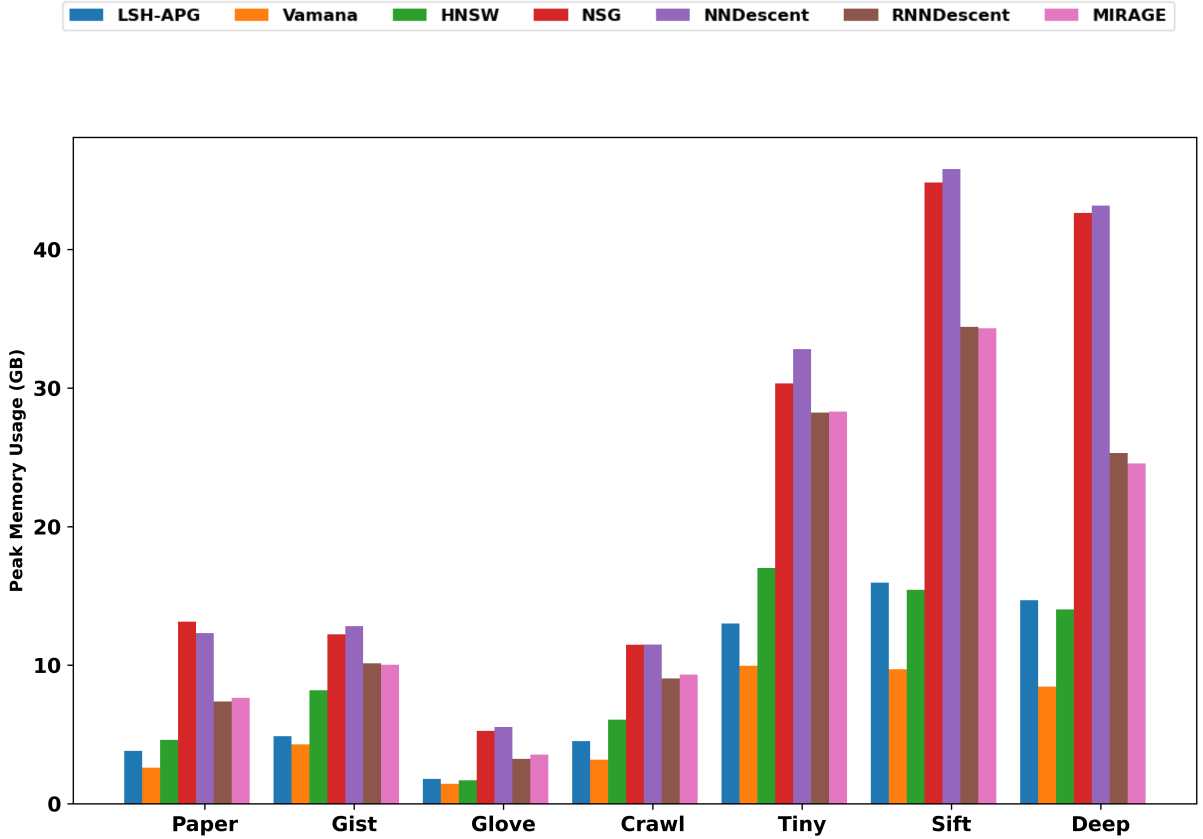


Figure 4.1: Comparison of peak memory usage for different methods

As can be seen in Figure 4.1, refinement-based approaches such as NSG and NN-Descent use the most memory to build the index, because they first build an approximate KNN graph. NSG uses NN-Descent to build a graph before pruning edges, and is therefore bound by its peak memory usage. Even though HNSW contains multiple layers, since it uses an increment-based approach starting with an empty graph, it does not build a K-Graph and therefore has a lower maximum out-degree than refinement-based approaches during construction. Consequently, it has lower memory use during construction. RNNDescent does not build a K-Graph either, but prunes during the process of building the graph (instead of post-pruning), which leads to its lower memory use compared to other refinement-based

approaches.

MIRAGE's memory overhead is dominated by its bottom layer where it also prunes while refining the graph. Upper layers are generated using an increment-based approach and do not affect memory consumption much.

4.3 Index Construction Performance

We evaluate the index construction efficiency of all the methods in terms of construction time. The results are shown in Figure 4.2.

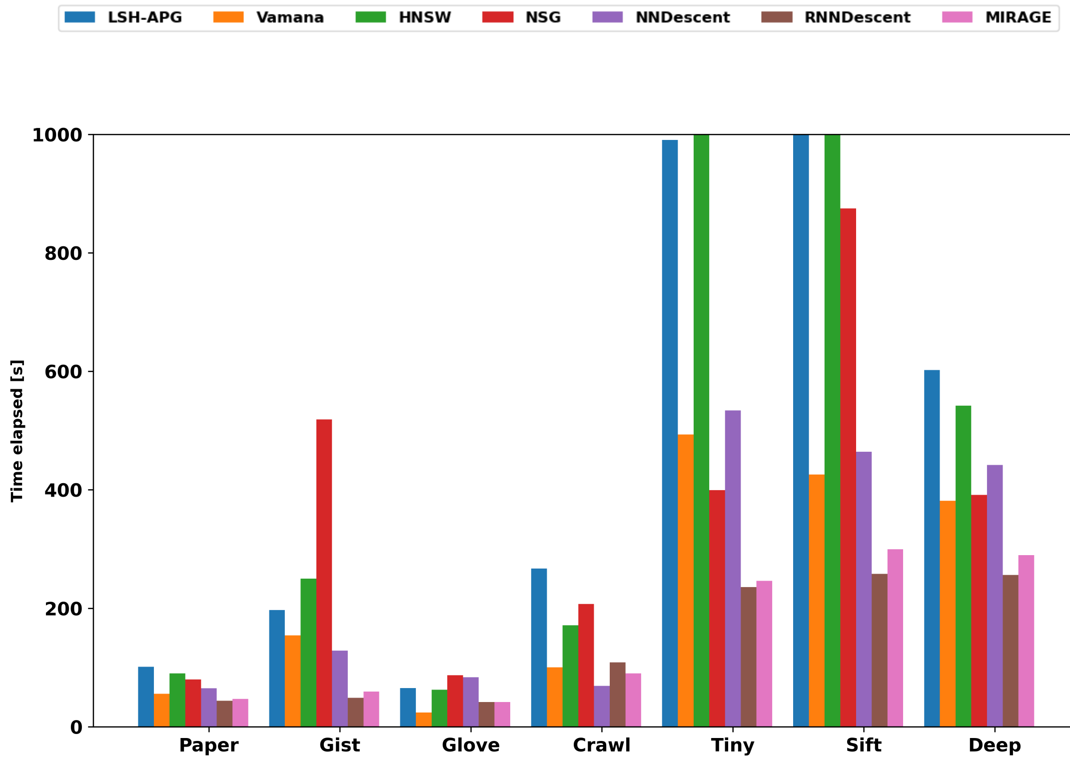


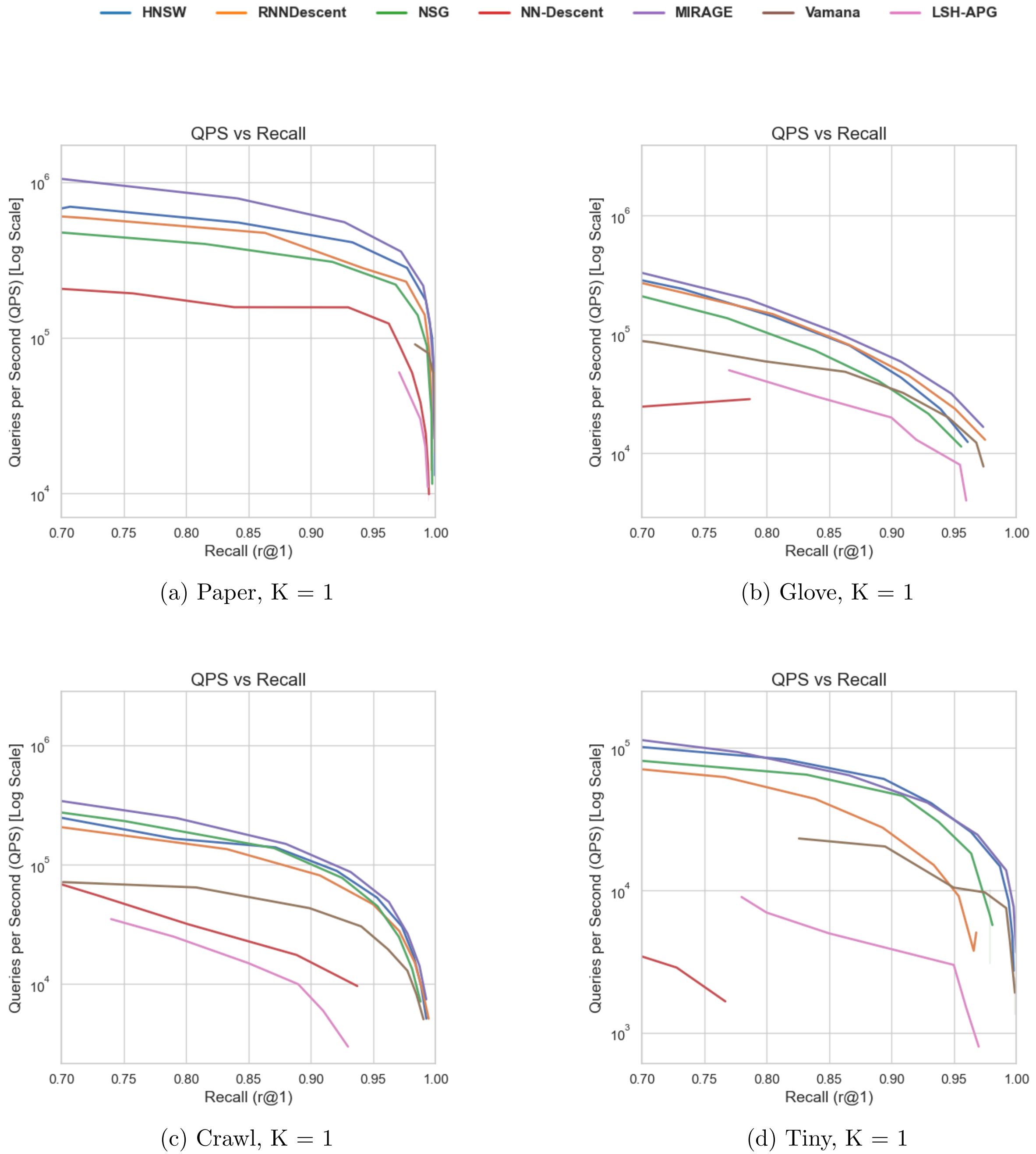
Figure 4.2: Indexing time of different graph-based methods. We cap the y-axis at 1000s for readability reasons. For the Tiny5M dataset, HNSW takes 1856.485s and for Sift10M HNSW takes 1051.475s and LSH-APG takes 1249.135s

Depending on the dataset, NN-Descent or RNNDescent constructs their indexes fastest, with MIRAGE taking slightly more time than RNNDescent to construct. When the size of the dataset is small, the slowdown is marginal, and even when the dataset size increases, it is only 20% slower at its worst. The slowdown is due to the hierarchical structure of MIRAGE in which Layers 1 to N require computing ANNS to construct. However, since the layers are only constructed on a subset of the total dataset, the slowdown is negligible. Vamana performs well on 2 datasets of 1M vectors, but is slow to construct as the number of vectors increases due to its high out degree and using ANNS to form edges for each point in the dataset. Similarly, HNSW uses ANNS to construct the index, causing relatively high construction times. MIRAGE constructs its index faster than HNSW and NSG on all benchmarked datasets – up to 6× faster than HNSW and up to 7× faster than NSG, depending on the dataset. MIRAGE constructs its index faster than NSG due to the fact NSG must create a KNN graph using NN-Descent first before pruning instead of pruning during the construction process like Layer 0 of MIRAGE. LSH-APG is significantly slower than MIRAGE on all datasets (as much as 6×).

4.4 Search Performance

For each dataset, we plot QPS vs recall to show the search performance of MIRAGE compared to the benchmarked methods (Figures 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 4.10). This requires adjusting search parameters for each method, namely the value of C in Algorithm 1. We search the graphs with C set to $(1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024)$. Generally, the higher the value of C, the higher the recall and the lower the QPS. A higher QPS generally shows higher search efficiency and is related to the number of distance computations that a search performs [63]. We look at different k values for each dataset and method, namely $k = 1$, $k = 10$, $k = 20$, and $k = 100$. We plot the recall on each dataset from 0.7 to 1; including full range does not provide more information and makes it hard to read.

NN-Descent does not have competitive recall performance as compared to the other solutions on many datasets, due to the issues we mentioned above with K-Graph. Both LSH-APG and Vamana have competitive recall performance but are orders of magnitude worse than the top performing solutions in terms of QPS, similar to the findings in previous work [67]. RNNDescent performs worse as the value of k increases; the original paper includes search performance only at $k = 1$ [40]. Previous work has noted that the performance of graph-based ANNS algorithms can vary wildly depending on the dataset based on their characteristics [63]. Irrespective, on all tested datasets, MIRAGE shows the



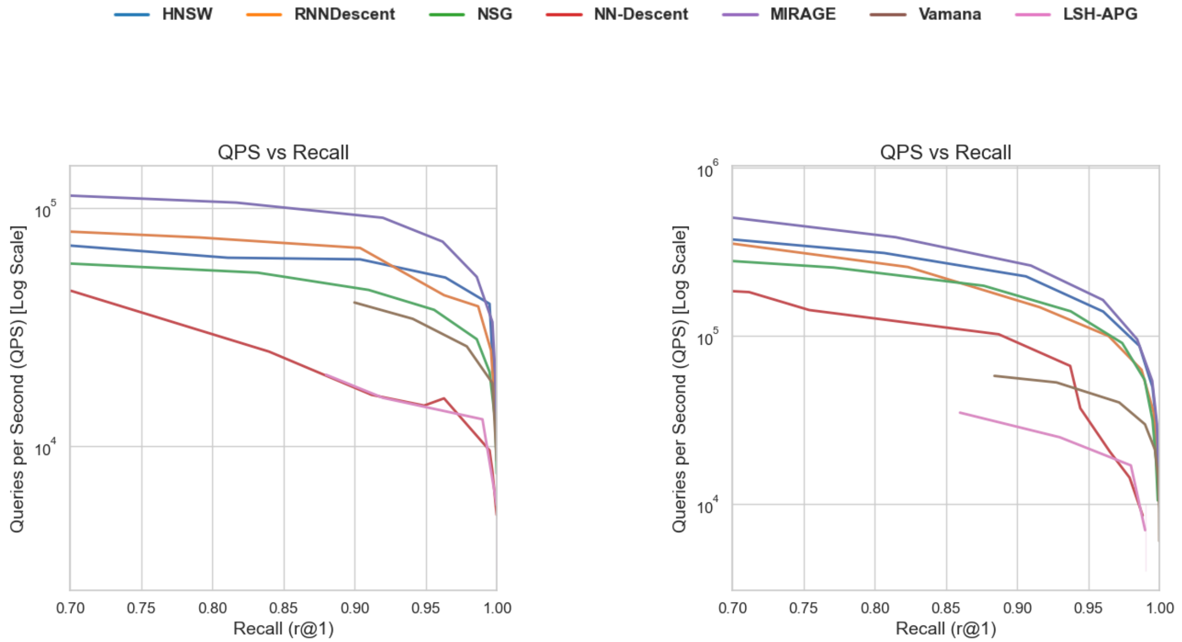
(a) Paper, K = 1

(b) Glove, K = 1

(c) Crawl, K = 1

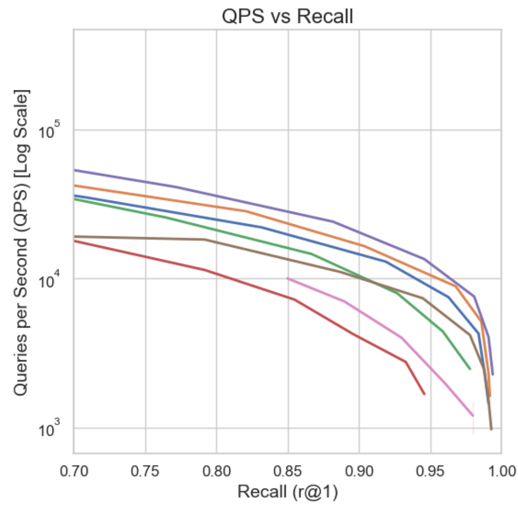
(d) Tiny, K = 1

Figure 4.3: QPS (log scale) vs recall where $k = 1$.



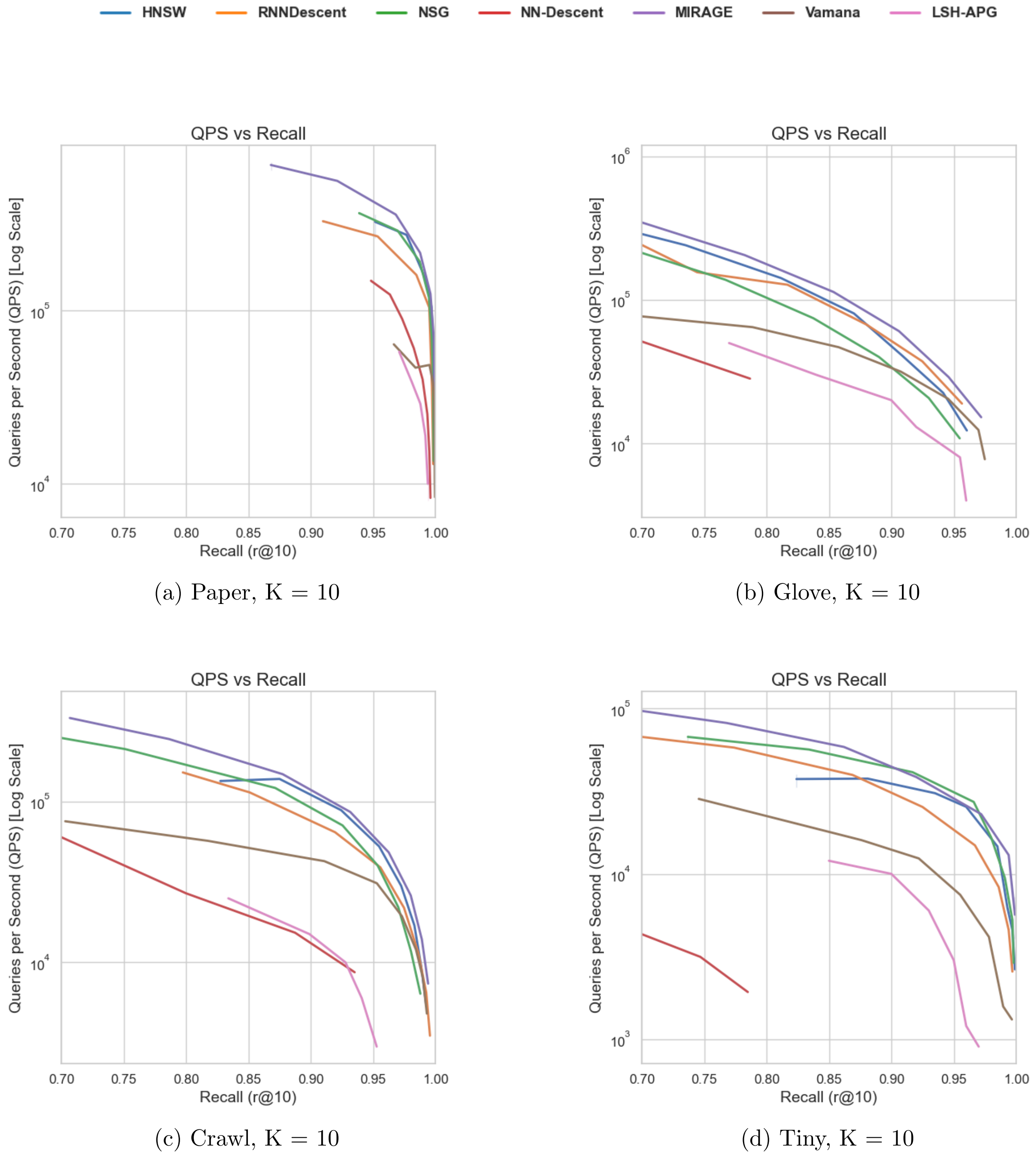
(a) Sift, K = 1

(b) Deep, K = 1



(c) Gist, K = 1

Figure 4.4: QPS (log scale) vs recall where $k = 1$.



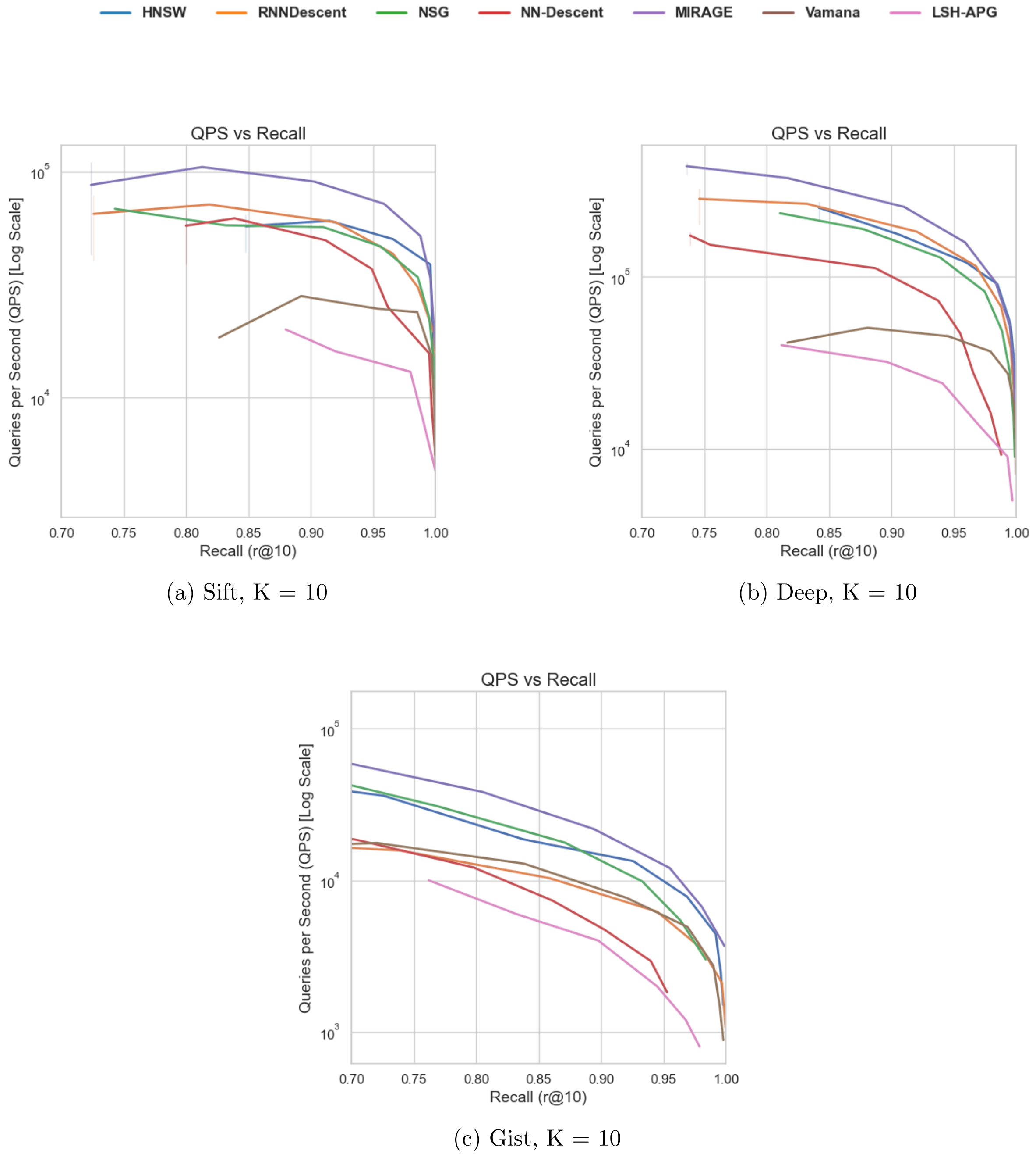
(a) Paper, K = 10

(b) Glove, K = 10

(c) Crawl, K = 10

(d) Tiny, K = 10

Figure 4.5: QPS (log scale) vs recall where $k = 10$.

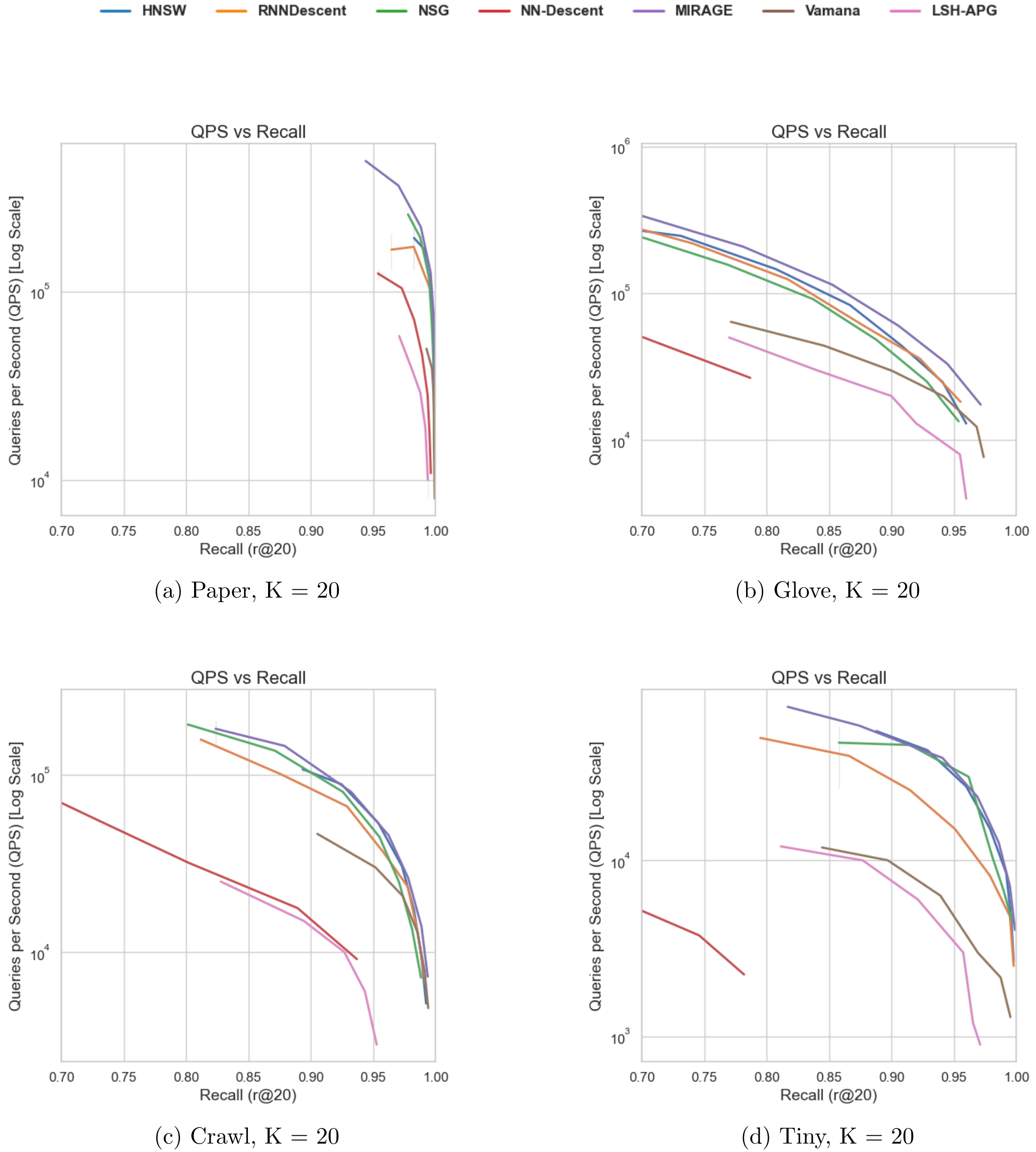


(a) Sift, K = 10

(b) Deep, K = 10

(c) Gist, K = 10

Figure 4.6: QPS (log scale) vs recall where $k = 10$.



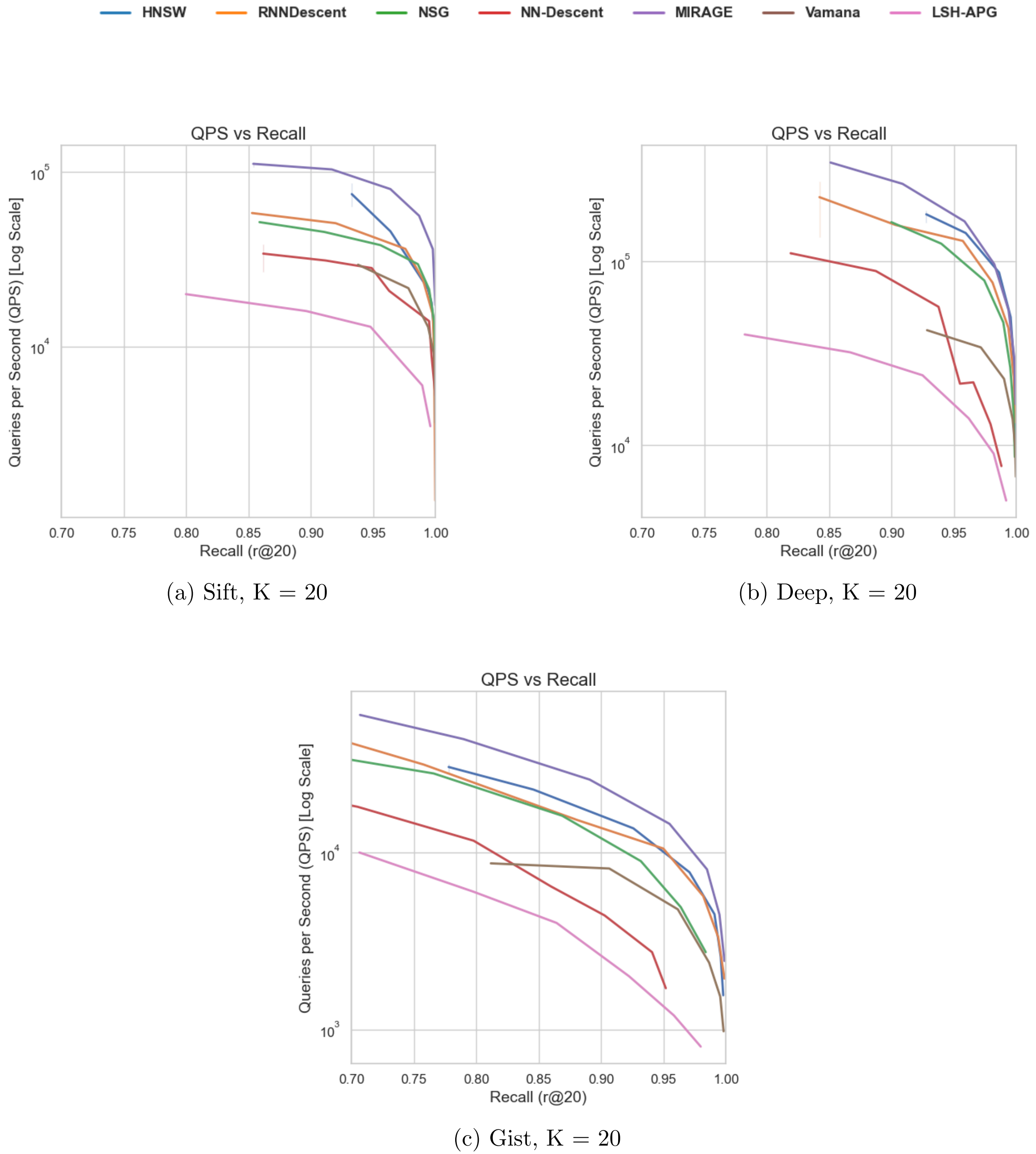
(a) Paper, K = 20

(b) Glove, K = 20

(c) Crawl, K = 20

(d) Tiny, K = 20

Figure 4.7: QPS (log scale) vs recall where $k = 20$.

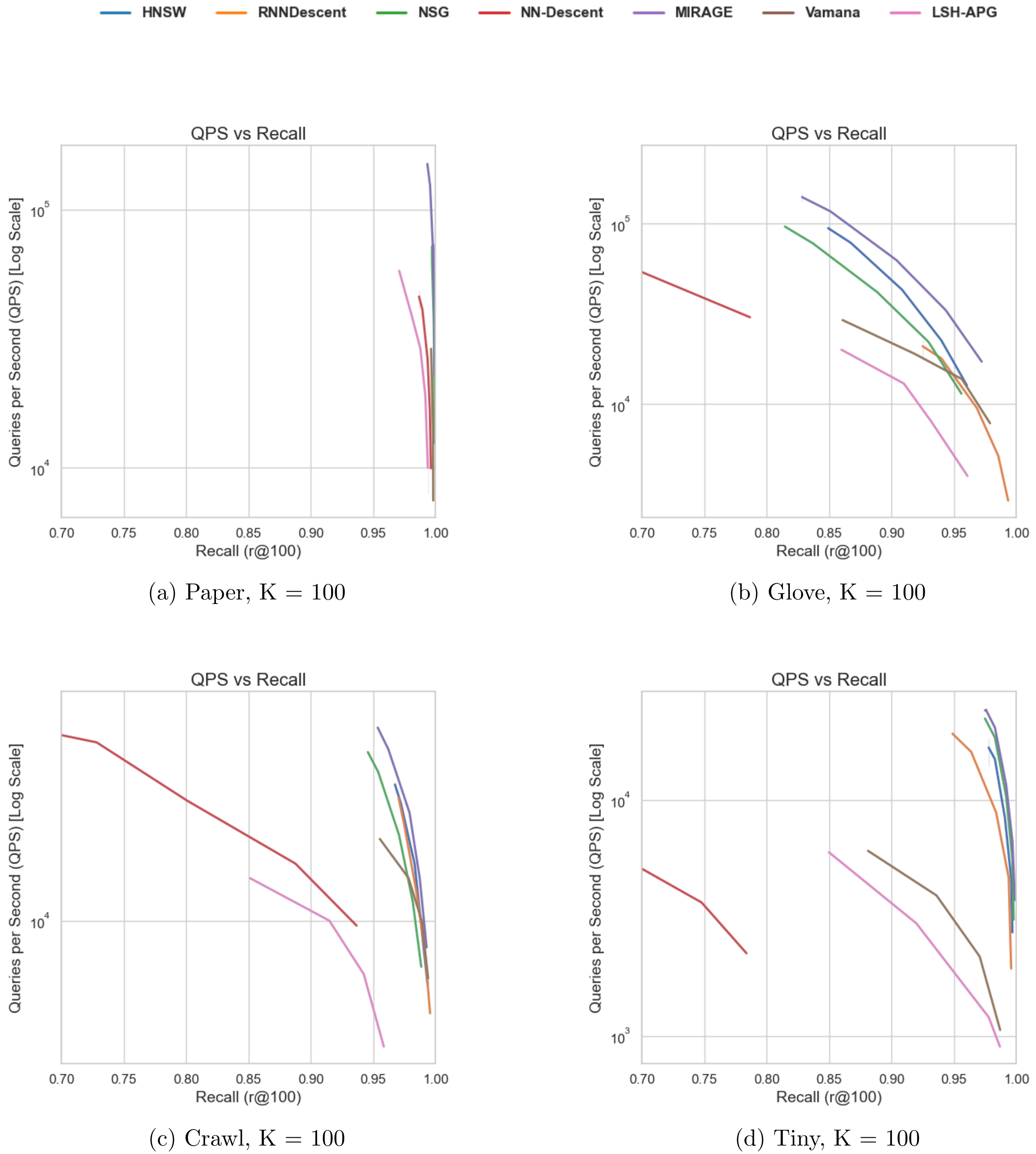


(a) Sift, K = 20

(b) Deep, K = 20

(c) Gist, K = 20

Figure 4.8: QPS (log scale) vs recall where $k = 20$.



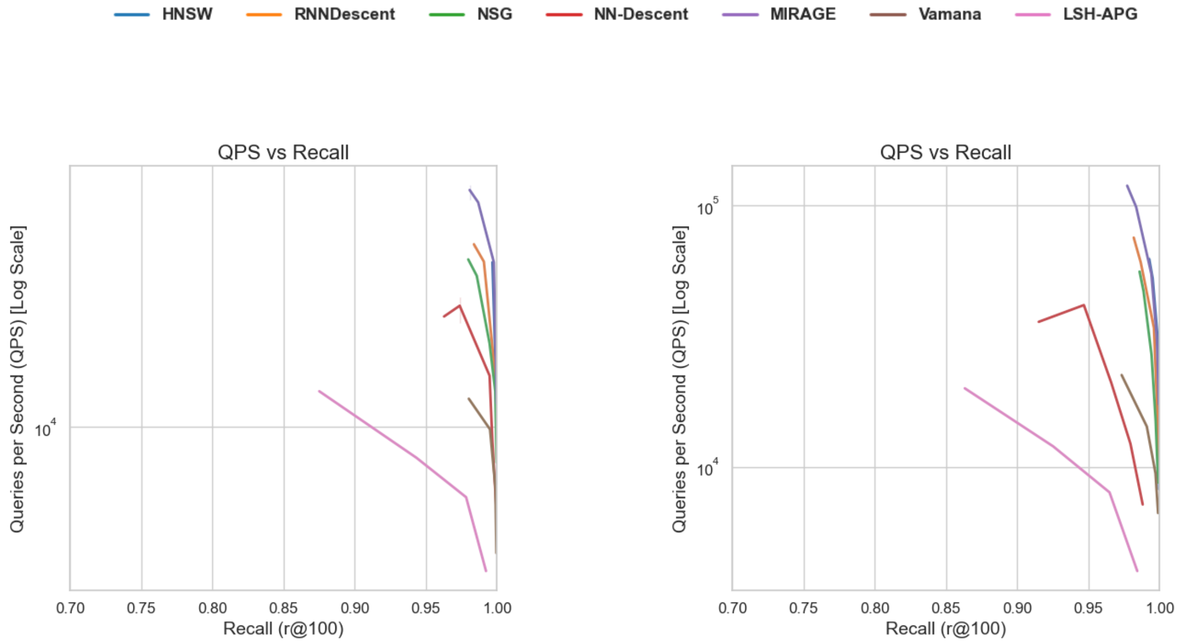
(a) Paper, K = 100

(b) Glove, K = 100

(c) Crawl, K = 100

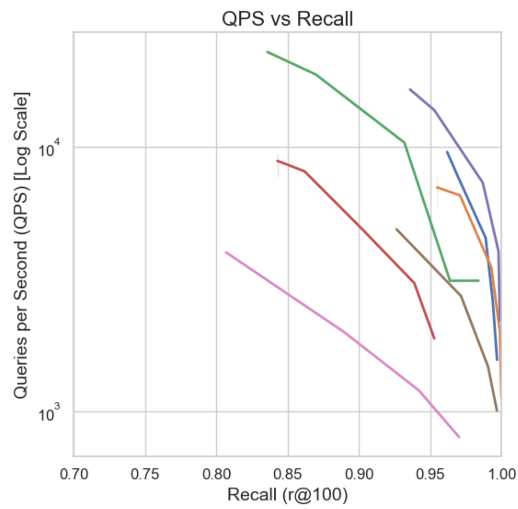
(d) Tiny, K = 100

Figure 4.9: QPS (log scale) vs recall where $k = 100$.



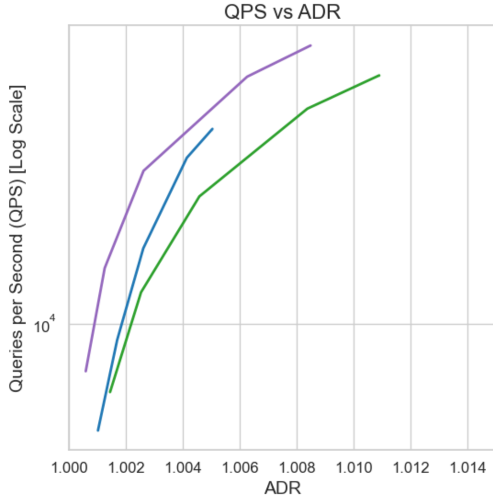
(a) Sift, K = 100

(b) Deep, K = 100

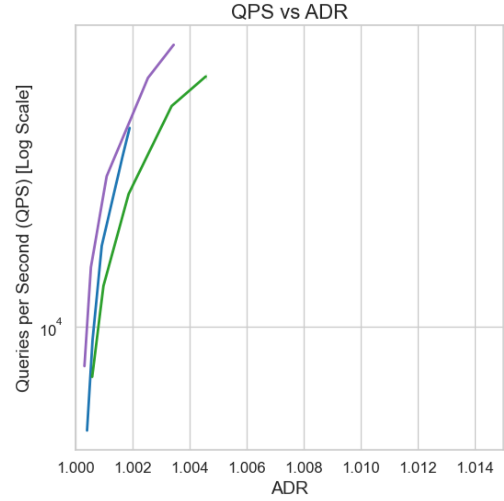


(c) Gist, K = 100

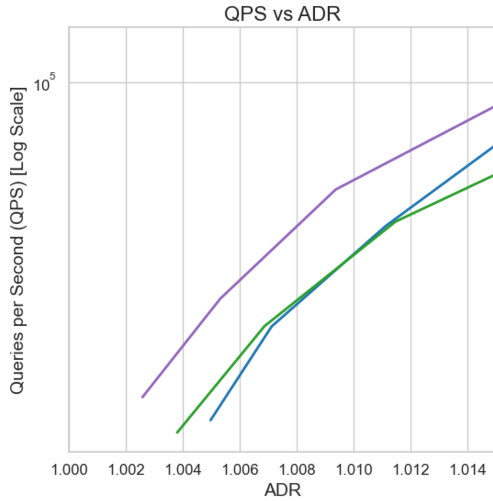
Figure 4.10: QPS (log scale) vs recall where $k = 100$.



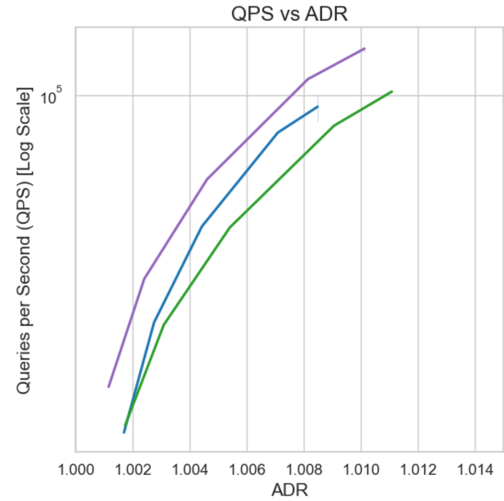
(a) Crawl, K = 1



(b) Crawl, K = 10



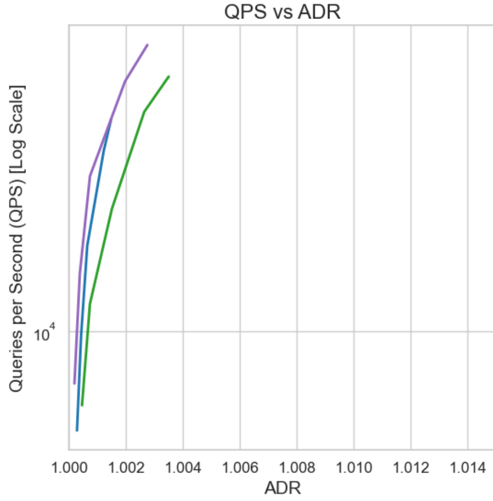
(c) Glove, K = 1



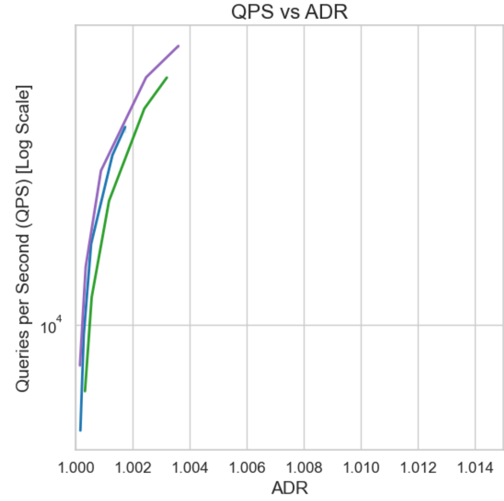
(d) Glove, K = 10

Figure 4.11: QPS (log scale) vs ADR of the Glove and Crawl datasets where k = 1 and k = 10.

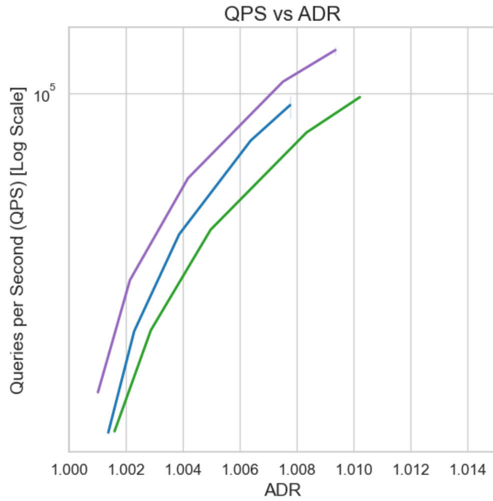
HNSW NSG MIRAGE



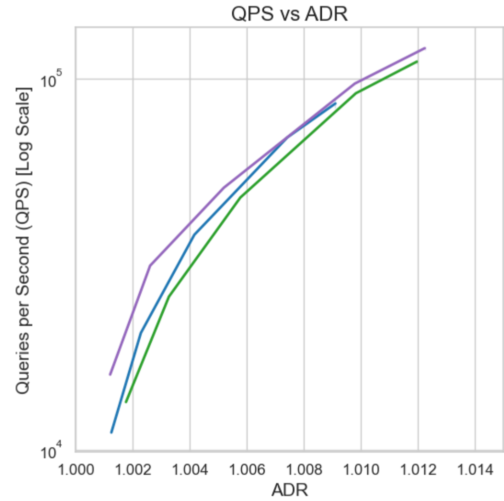
(a) Crawl, K = 20



(b) Crawl, K = 100



(c) Glove, K = 20



(d) Glove, K = 100

Figure 4.12: QPS (log scale) vs ADR of the Glove and Crawl datasets where $k = 20$ and $k = 100$.

highest QPS at the highest recalls, performing up to 2× better than HNSW and NSG at the highest recalls.

In order to have a more fine grained look at search accuracy, we take the top three performing algorithms in terms of search (MIRAGE, NSG, and HNSW) and plot QPS vs ADR for two datasets with high LIDs (Glove and Crawl) and all the k values mentioned above. These are shown in Figures 4.11 and 4.12. At all k values on both datasets, MIRAGE has the highest QPS while also having the lowest ADR, confirming it is the best solution both in terms of search accuracy and throughput.

Although MIRAGE constructs its index faster than HNSW and NSG, it is still able to have better search efficiency than both solutions at the same or better accuracy. The following sections discuss experiments that explain the reasons for this.

4.5 Graph Qualities

As mentioned in Section 4.2, the index size is mostly a function of the average out-degree of the graph [63]. Thus we consider average out-degree and index size for each graph generated from all 7 datasets (Table 4.2). NN-Descent produces an index with the highest average out-degree out of all methods considered, and therefore has the largest index size. This is because it does not prune edges. Vamana has the second highest average out-degree on all datasets. Depending on the dataset, HNSW (Layer 0) or LSH-APG have the third highest average out-degree, and RNNDescent or NSG have the fourth-highest average out-degree. MIRAGE has a lower average out-degree than RNNDescent due to its maximum out-degree restriction. However, MIRAGE has to store the upper layers of the graph. Since the layers only contain a subset of the total dataset, this is not much extra overhead, around 0.5GB extra on the Tiny dataset. In terms of search performance, the search cost of the greedy search algorithm is the product of the length of the search path and the out-degree of each point in the path [64]. The low average out-degree of MIRAGE allows it to perform fewer distance computations compared to other solutions, helping it maintain its superior search performance across datasets. We expand on this more in Section 4.8.

4.6 Parameter Experiments

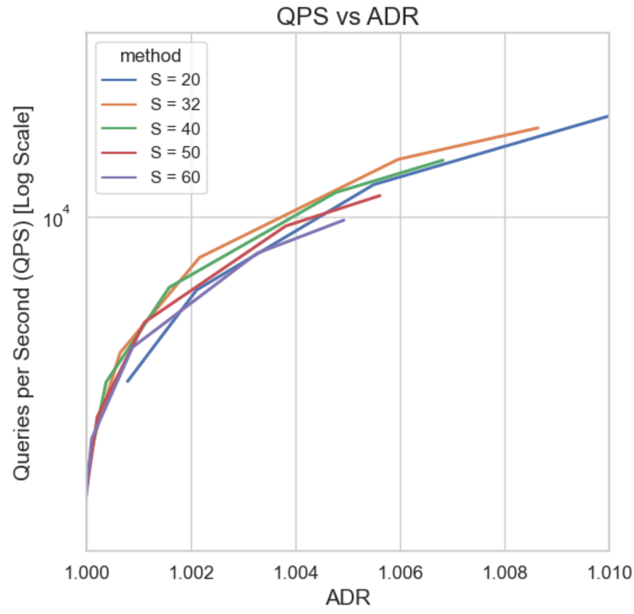
We study the effects of the construction parameters on both construction and search performance. For construction we look at the construction time and for search we look at the

Graph	Gist		Glove		Crawl		Paper		Sift		Tiny		Deep	
	AOD	IS	AOD	IS	AOD	IS	AOD	IS	AOD	IS	AOD	IS	AOD	IS
HNSW Layer 0	21.81839	3.658	15.892	0.511	13.786	2.326	26.834	1.715	43.678	6.396	26.946	7.654	46.512	5.307
MIRAGE Layer 0	17.424	3.641	15.622	0.510	10.084	2.299	11.036	1.596	24.917	5.697	21.272	7.549	25.145	4.512
NSG	12.531	3.623	9.599	0.483	10.289	2.300	18.612	1.653	27.972	5.810	18.171	7.491	32.825	4.798
RNNDescent	22.127	3.659	20.743	0.532	11.698	2.311	12.955	1.610	26.838	5.768	32.809	7.764	27.536	4.601
NN-Descent	100	3.949	100	0.882	80	2.817	100	2.269	90	8.121	100	9.015	90	6.926
Vamana	67.225	3.83	56.427	0.706	39.754	2.579	66.194	2.061	69.423	7.531	69.283	8.645	69.305	6.35
LSH-APG	32.443	3.785	29.630	0.585	31.059	2.513	35.262	1.821	36.610	6.279	32.990	7.953	35.666	5.022

Table 4.2: Average vertex out-degree (AOD) and Index size (IS) for each method. Index size is represented in GB.

QPS vs ADR. All parameter experiments are run on the Gist dataset, and we set $k = 100$ for all search experiments. We choose $k = 100$, since when running initial experiments we observed recall degradation at this k value, prompting us to test the impact of graph density/construction parameters on search performance at $k = 100$ value. We choose ADR to get a more fine-grained look at search accuracy performance.

First we look at S, which represents the initial out-degree of the randomly connected graph at Layer 0 before refinement. As Figure 4.13 depicts, increasing S causes construction time to increase. This is due to a denser graph being constructed, with each vertex having more neighbours. We see a similar phenomenon in search, where there is a direct relationship between the graph density and the QPS vs ADR trade off. A denser graph (higher value of S) has lower QPS and has less accuracy degradation. The decrease in QPS is due to more distance computations being done, related to the average out-degree of a denser graph. Empirically we find the best search performance in terms of QPS and accuracy when $S = 32$ on most datasets tested; even when $S = 60$, MIRAGE is constructed $2.6\times$ faster than HNSW and $5\times$ faster than NSG.



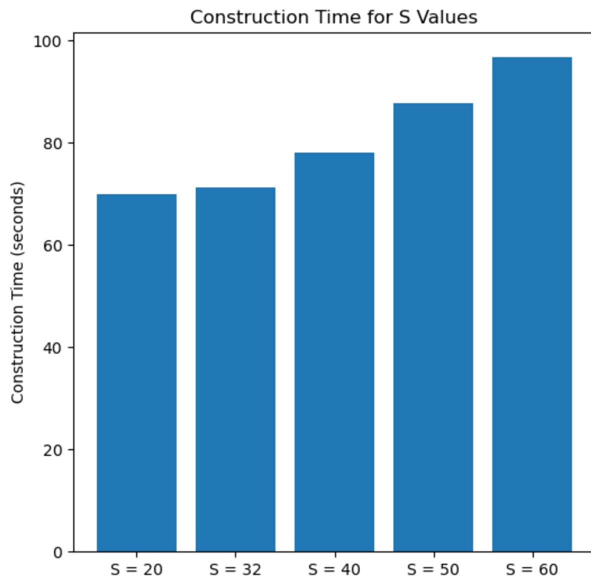


Figure 4.13: Effect of different S values on construction time and search performance (QPS vs ADR)

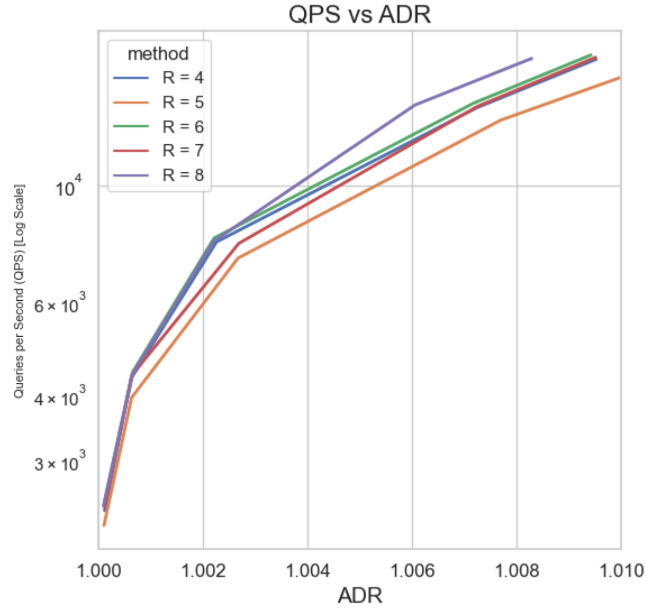
Next we look at R, which represents the number of iterations of reverse edge additions. We fix the value of $Iter$, the number of refinement iterations to 15. The results, shown in Figure 4.14, reveal that an increased R value improves search performance and also increases construction time. This illustrates the importance of reverse edge addition to the Layer 0 of MIRAGE for optimal search performance.

Lastly, we look at $Iter$, which represents the number of refinement iterations MIRAGE goes through at Layer 0. We fix $R = 4$ in these experiments. The results in Figure 4.15 show there is a negligible difference in construction time between all values, with higher values of $Iter$ yielding marginally higher construction times. The main insight here is that after a certain amount of refinement iterations, the improvement in search becomes negligible and can even make the search perform worse due to insufficient opportunities to add reverse edges at a fixed R value. Due to this, we find a less drastic difference between R and $Iter$ to have the best trade off between construction and search performance, for example $R = 5$ and $Iter = 12$ instead of $R = 4$ and $Iter = 50$.

Overall, we find that having a denser graph at Layer 0 can help reduce search accuracy degradation at lower recalls, with the trade-off of reduced QPS. Depending on the application needs in terms of search accuracy and throughput, MIRAGE can be configured for best search performance in both scenarios.

4.7 Ablation Study

We investigate the hierarchical search component of MIRAGE in ablation studies to determine its effectiveness. To test the benefits of the hierarchy, we construct MIRAGE with and without hierarchy, and search both structures. We keep the average out-degree of Layer 0 of hierarchical MIRAGE the same as the version without hierarchy for a fair comparison. We note that this is the worst-case scenario for MIRAGE since its benefits come from reducing the density of its bottom layer and getting assistance from the hierarchy to navigate to a good entry point and quickly search the lower layer. For searching, we find random entry points in the non-hierarchical graph, as they do not have entry points from upper layers, and for the hierarchical version of MIRAGE we use our standard search algorithm. Figure 4.16 shows the results on the Gist and Paper datasets. The results show that even when the out-degree is the same, the hierarchical solution is able to achieve better search performance due to providing better entry points to the bottom layer than randomly choosing a start point. This is due to the fact that the upper layers are constructed using an increment-based approach, creating a higher quality graph.



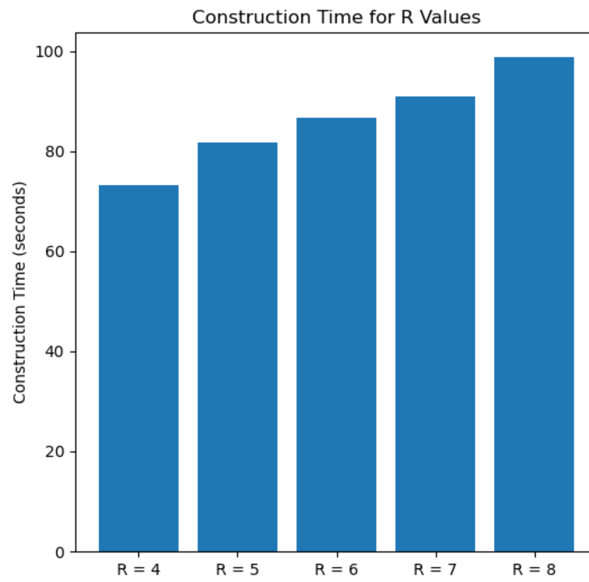
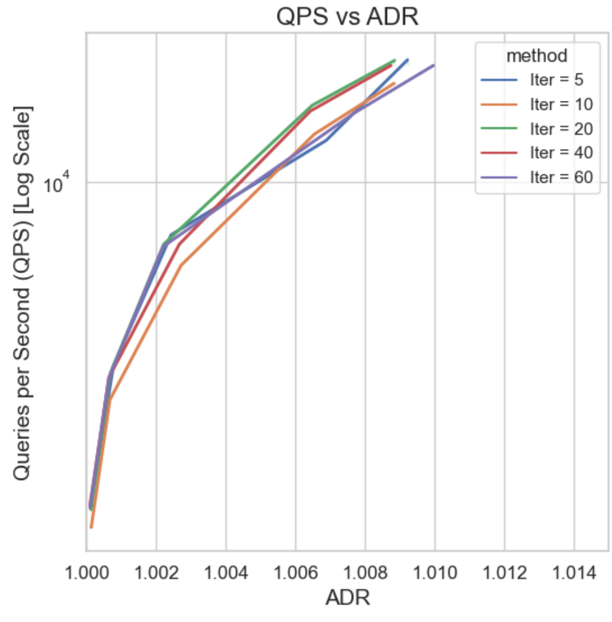


Figure 4.14: Effect of different R values on construction time and search performance (QPS vs ADR)



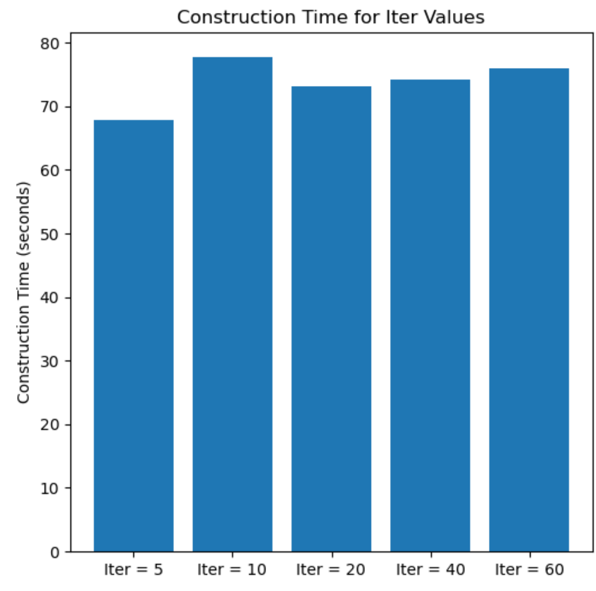


Figure 4.15: Effect of different Iter values on construction time and search performance (QPS vs ADR)

Table 4.3: Number of distance computations to achieve 0.99 recall with $k = 100$. Number in parenthesis represents the difference between HNSW and MIRAGE.

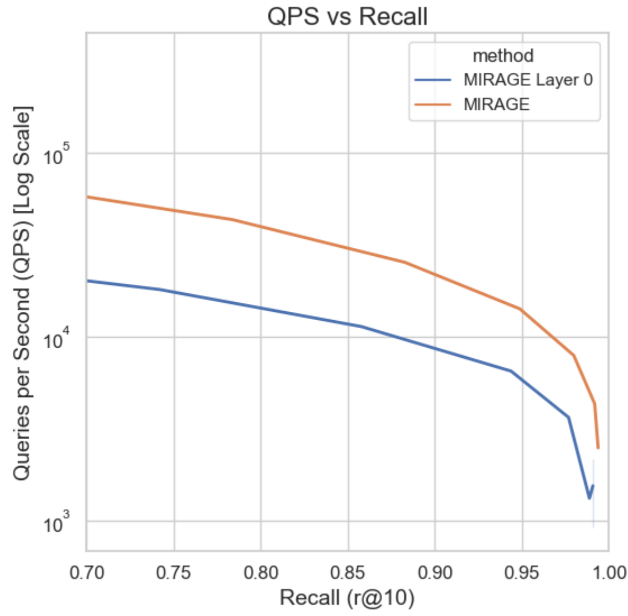
	HNSW	MIRAGE
Paper	3936 (2.4× greater)	1588
Crawl	4319 (2.4× greater)	1758
Gist	25161 (1.7× greater)	14776
Sift	1474 (2× greater)	716

4.8 Discussion

In this section we discuss the reasons why MIRAGE performs better than other methods in terms of search performance; we already discussed why it performs well in terms of index construction time in Sections 3.2 and 4.3. Recall that the search cost of the greedy search algorithm is the product of the length of the search path and the out-degree of each point in the path [64]. The out-degree of the points determines the number of distance computations that a method needs to perform.

Table 4.3 shows the number of distance computations needed to achieve 0.99 recall when $k = 100$ for MIRAGE and HNSW – a meaningful comparison since they both use a hierarchical greedy search. The results show that MIRAGE performs around 2× fewer distance comparisons than HNSW during the search process. This is what leads to MIRAGE's superior QPS. The reason for the fewer computations is because MIRAGE has a lower average out-degree than HNSW at Layer 0. In addition, maintaining a sparse lower layer reduces the chances of MIRAGE getting caught in local optima, since too many local connections in a graph lead to more local optima, which may hinder the search accuracy [64].

As shown in the ablation study, adding hierarchy improves the search performance of MIRAGE by having longer edges that can traverse the whole graph quickly and provide good entry points for the lower layer. There is existing work suggesting that HNSW does not benefit that much from hierarchy due to a small subset of well-connected nodes (hubs) naturally forming a "highway" structure at Layer 0 [31, 39]. We find in our ablation study that hierarchy helps MIRAGE, likely due to the fact that Layer 0 and upper layers are constructed using different approaches – Layer 0 with a refinement-based approach and upper layers with a increment-based approach. This is different from HNSW, where all layers are constructed using the increment-based algorithm. MIRAGE benefits from having this "highway" structure in its upper layers in a way that HNSW perhaps does not.



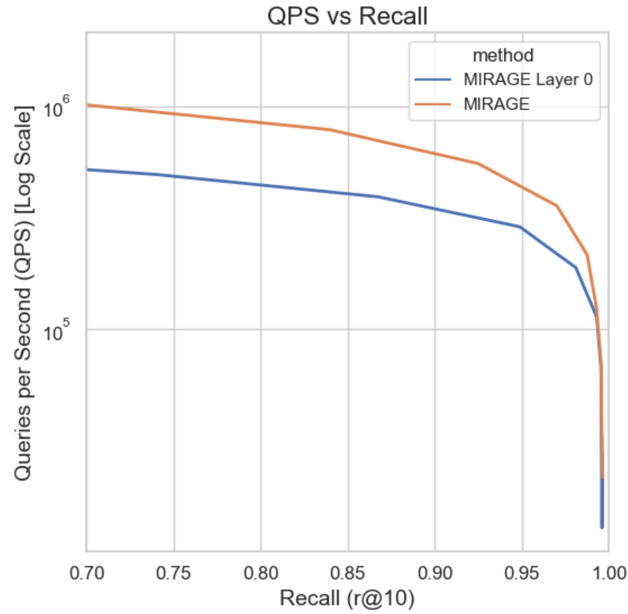


Figure 4.16: Ablation Study on MIRAGE Layer 0 vs Hierarchical MIRAGE . Gist dataset is shown on top, Paper dataset is shown on the bottom

Chapter 5

Conclusion

In this work, we present MIRAGE, a novel graph-based multilayered index for Approximate Nearest neighbour Search (ANNS) that combines the benefits of both refinement-based and increment-based approaches. Using a novel construction strategy that builds the bottom layer of the graph index using a refinement-based approach and higher layers using an increment-based approach, we ensure fast construction times while maintaining high search accuracy and supporting incremental inserts. Our extensive evaluation on multiple real-world datasets demonstrates that MIRAGE consistently outperforms state-of-the-art increment-based methods such as HNSW in terms of index construction and query recall, query speed, and refinement-based methods such as NSG, NN-Descent, and RNNDescent in terms of query speed and ability to perform incremental inserts without requiring full graph reconstruction. To be more specific, MIRAGE can be constructed 6× faster than HNSW and 7× faster than NSG, depending on the dataset, while also delivering up to 2× more QPS at a fixed recall. Through studying graph properties and with ablation studies, we show that each component of MIRAGE is providing a benefit, and demonstrate why MIRAGE performs so well across a range of datasets. These results indicate that MIRAGE is well-suited for large-scale ANNS tasks across a variety of applications.

We are following a number of directions for improving MIRAGE and using it in downstream tasks. One immediate issue is to incorporate the ability to handle vector deletions through incremental updates of the MIRAGE index. Vertex deletions in directed graphs is a difficult problem as the vertex is connected to other vertices and many edges need to be dropped [56]. Consequently, most in-memory graph-based indexes ignore deletions [33]. However, there are ways to incorporate them, perhaps by the use of auxiliary data structures [70] and this is an issue under consideration. We believe that there are ways to improve ANNS over MIRAGE performance further by the use of hardware accelerators

similar to the other graph-based methods mentioned [41, 69, 68]. GPU-based methods traditionally do not add hierarchy to their index since initial nodes can be randomly selected and compared to the query, thus employing the high parallelism and memory bandwidth of the GPU [41]. CAGRA [41] is constructed in a refinement-based approach on the GPU using NN-Descent and then pruned after, one path of exploration could be pruning during the process of construction, similar to how MIRAGE constructs Layer 0.

Lastly, given the superior performance of MIRAGE in both index construction and search, we are interested in using it as the basis of filtered query processing. These queries combine ANNS over vector space with predicates over structured data, and are crucial for applications that involve multiple data modalities, such as finding similar images while applying metadata-based filters. Pre-filtering and post-filtering are the main strategies for filtered queries. Pre-filtering first applies the structured predicate and then performs ANNS on the vector space. It becomes inefficient as selectivity increases or when dealing with large datasets. Post-filtering performs the ANNS first, then applies the filter, but often requires expanding the search scope, leading to inefficiencies for low-selectivity predicates.

Examples of graph-based solutions include Filtered-DiskANN [22], Milvus [60], ACORN [44], and iRangeGraph [65]. Efficiently executing filtered queries remains a challenge, requiring both fast query performance and query semantics to handle a diverse range of predicates that may not be predefined.

Filtered search solutions such as ACORN [44] can take a long time to construct due to their high average out degree, which is needed so that there are enough edges that pass any given filter. Using MIRAGE to construct a filtered search index could potentially have fast construction times while not compromising search performance.

References

[1] https://www.cse.cuhk.edu.hk/systems/hash/gqr/datasets.html. [Accessed 08-10-2024].

[2] ANNOY library. https://github.com/spotify/annoy. Accessed: 08-10-2024.

[3] Common Crawl - Open Repository of Web Crawl Data — commoncrawl.org. https://commoncrawl.org/. [Accessed 08-10-2024].

[4] Evaluation of Approximate nearest neighbors: large datasets — corpus-texmex.irisa.fr. http://corpus-texmex.irisa.fr/. [Accessed 08-10-2024].

[5] Gpt-4.

[6] Open-source vector similarity search for postgres.

[7] Pynndescent.

[8] The vector database to build knowledgeable AI — Pinecone — pinecone.io. https://www.pinecone.io/. [Accessed 14-10-2024].

[9] vespa.ai. https://vespa.ai/. [Accessed 14-10-2024].

[10] Ilias Azizi, Karima Echihabi, and Themis Palpanas. Graph-based vector search: An experimental evaluation of the state-of-the-art. *Proc. ACM Manag. Data*, 3(1), 2025.

[11] Brankica Bratić, Michael E. Houle, Vladimir Kurbalija, Vincent Oria, and Miloš Radovanović. NN-Descent on high-dimensional data. In *Proc. 8th Intl. Conf. on Web Intelligence, Mining and Semantics*. Association for Computing Machinery, 2018.

[12] Qi Chen, Haidong Wang, Mingqin Li, Gang Ren, Scarlett Li, Jeffery Zhu, Jason Li, Chuanjie Liu, Lintao Zhang, and Jingdong Wang. Sptag: A library for fast approximate nearest neighbor search. https://github.com/Microsoft/SPTAG, 2018.

[13] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. Spann: Highly-efficient billion-scale approximate nearest neighborhood search. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 5199–5212. Curran Associates, Inc., 2021.

[14] Felix Chern, Blake Hechtman, Andy Davis, Ruiqi Guo, David Majnemer, and Sanjiv Kumar. TPU-KNN: k nearest neighbor search at peak FLOP/s. In *Proc. 36th Intl. Conf. on Neural Information Processing Systems*, NIPS '22. Curran Associates Inc., 2024.

[15] Wei Dong, Charikar Moses, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proc. 20th Intl. Conf. on World Wide Web*, pages 577–586. Association for Computing Machinery, 2011.

[16] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library. https://github.com/facebookresearch/faiss, 2024.

[17] Cong Fu and Deng Cai. EFANNA : An extremely fast approximate nearest neighbor search algorithm based on kNN graph. arXiv 1609.07228, 2016.

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

[19] Junhao Gan, Jianlin Feng, Qiong Fang, and Wilfred Ng. Locality-sensitive hashing scheme based on dynamic collision counting. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 541–552. Association for Computing Machinery, 2012.

[20] Jianyang Gao and Cheng Long. High-dimensional approximate nearest neighbor search: with reliable and efficient distance comparison operations. *Proc. ACM Manag. Data*, 1(2), June 2023.

[21] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *Proc. 25th Intl. Conf. on Very Large Data Bases*, pages 518–529. VLDB Endowment, 1999.

[22] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar

Srinivasan, Amit Singh, and Harsha Vardhan Simhadri. Filtered-diskann: Graph algorithms for approximate nearest neighbor search with filters. In *Proc. ACM Web Conf. 2023*, pages 3406–3416. Association for Computing Machinery, 2023.

[23] Fabian Groh, Lukas Ruppert, Patrick Wieschollek, and Hendrik P. A. Lensch. GGNN: Graph-based GPU nearest neighbor search. *IEEE Trans on Big Data*, 9(1):267–279, 2023.

[24] Kiana Hajebi, Yasin Abbasi-Yadkori, Hossein Shahbazi, and Hong Zhang. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *Proc. 22nd Intl. Joint Conf. on Artificial Intelligence*, pages 1312–1317, 2011.

[25] Nico Hezel, Kai Uwe Barthel, Konstantin Schall, and Klaus Jung. An exploration graph with continuous refinement for efficient multimedia retrieval. In *Proc. of 2024 International Conference on Multimedia Retrieval*, pages 657–665. Association for Computing Machinery, 2024.

[26] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *Proc. VLDB Endow.*, 9(1):1–12, September 2015.

[27] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proc. Thirtieth Annual ACM Symposium on Theory of Computing*, pages 604–613. Association for Computing Machinery, 1998.

[28] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proc. 44th Annual*

Intl. Symposium on Computer Architecture, pages 1–12. Association for Computing Machinery, 2017.

[29] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. Approximate nearest neighbor search on high dimensional data — experiments, analyses, and improvement. *IEEE Trans. on Knowledge and Data Eng.*, 32(8):1475–1488, 2020.

[30] Shengwen Liang, Ying Wang, Ziming Yuan, Cheng Liu, Huawei Li, and Xiaowei Li. Vstore: in-storage graph based vector search accelerator. In *Proc. 59th ACM/IEEE Design Automation Conf.*, pages 997–1002. Association for Computing Machinery, 2022.

[31] Peng-Cheng Lin and Wan-Lei Zhao. Graph based nearest neighbor search: Promises and failures. https://arxiv.org/abs/1904.02077, 2019.

[32] Kejing Lu, Yoshiharu Ishikawa, and Chuan Xiao. MQH: Locality sensitive hashing on multi-level quantization errors for point-to-hyperplane distances. *Proc. VLDB Endow.*, 16(4):864–876, 2022.

[33] Kejing Lu, Mineichi Kudo, Chuan Xiao, and Yoshiharu Ishikawa. HVS: hierarchical graph structure based on voronoi diagrams for solving approximate nearest neighbor search. *Proc. VLDB Endow.*, 15(2):246–258, 2021.

[34] Yu A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 42(4):824–836, 2020.

[35] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Inf. Syst.*, 45:61–68, 2014.

[36] Magdalen Dobson Manohar, Zheqi Shen, Guy Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. ParlayANN: Scalable and deterministic parallel graph-based approximate nearest neighbor search algorithms. In *Proc. 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 270–285. Association for Computing Machinery, 2024.

[37] Yusuke Matsui, Yusuke Uchida, Hervé Jégou, and Shin'ichi Satoh. A survey of product quantization. *ITE Trans. on Media Technology and Applications*, 6(1):2–10, 2018.

[38] Jason Mohoney, Anil Pacaci, Shihabur Rahman Chowdhury, Ali Mousavi, Ihab F. Ilyas, Umar Farooq Minhas, Jeffrey Pound, and Theodoros Rekatsinas. High-throughput vector similarity search in knowledge graphs. *Proc. ACM Manag. Data*, 1(2), June 2023.

[39] Blaise Munyampirwa, Vihan Lakshman, and Benjamin Coleman. Down with the hierarchy: The 'h' in hnsw stands for "hubs". https://arxiv.org/abs/2412.01940, 2024.

[40] Naoki Ono and Yusuke Matsui. Relative NN-Descent: A fast index construction for graph-based approximate nearest neighbor search. In *Proc. 31st ACM Intl. Conf. on Multimedia*, pages 1659–1667. Association for Computing Machinery, 2023.

[41] H. Ootomo, A. Naruse, C. Nolet, R. Wang, T. Feher, and Y. Wang. CAGRA: Highly parallel graph construction and approximate nearest neighbor search for GPUs. In *Proc. 40th IEEE Intl. Conf. on Data Engineering*, pages 4236–4247. IEEE Computer Society, 2024.

[42] James Jie Pan, Jianguo Wang, and Guoliang Li. Survey of vector database management systems. https://arxiv.org/abs/2310.14021, 2023.

[43] James Jie Pan, Jianguo Wang, and Guoliang Li. Vector database management techniques and systems. In *Companion of the 2024 International Conference on Management of Data*, pages 597–604. Association for Computing Machinery, 2024.

[44] Liana Patel, Peter Kraft, Carlos Guestrin, and Matei Zaharia. Acorn: Performant and predicate-agnostic search over vector embeddings and structured data. *Proc. ACM Manag. Data*, 2(3), May 2024.

[45] Marco Patella and Paolo Ciaccia. The many facets of approximate similarity search. In *Proc. First International Workshop on Similarity Search and Applications*, pages 10–21, 2008.

[46] Marco Patella and Paolo Ciaccia. Approximate similarity search: A multi-faceted problem. *J. of Discrete Algorithms*, 7(1):36–48, March 2009.

[47] Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global vectors for word representation. In Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors, *Proc. 2014 Conf. on Empirical Methods in Natural Language Processing*, pages 1532–1543. Association for Computational Linguistics, October 2014.

[48] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, June 1990.

[49] M M Mahabubur Rahman and Jelena Tešić. Stratified graph indexing for efficient search in deep descriptor databases. *Int. J. Multimed. Inf. Retr.*, 13(3), 2024.

[50] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. Freshdiskann: A fast and accurate graph-based ann index for streaming similarity search. https://arxiv.org/abs/2105.09613, 2021.

[51] Martin Skrodzki. The k-d tree data structure and a proof for neighborhood computation in expected logarithmic time. https://arxiv.org/abs/1903.04936, 2019.

[52] Suhas Jayaram Subramanya, Devvrit, Rohan Kadekodi, Ravishankar Krishaswamy, and Harsha Vardhan Simhadri. Diskann: fast accurate billion-point nearest neighbor search on a single node. In *Proc. of the 33rd International Conference on Neural Information Processing Systems*. Curran Associates Inc., 2019.

[53] Yifang Sun, Wei Wang, Jianbin Qin, Ying Zhang, and Xuemin Lin. Srs: solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index. *Proc. VLDB Endow.*, 8(1):1–12, September 2014.

[54] Eric S. Tellez, Martin Aumüller, and Edgar Chavez. Overview of the sisap 2023 indexing challenge. In *Similarity Search and Applications: 16th International Conference*, pages 255–264. Springer-Verlag, 2023.

[55] Eric S. Tellez, Martin Aumüller, and Vladimir Mic. Overview of the sisap 2024 indexing challenge. In *Similarity Search and Applications: 17th International Conference*, pages 255–265. Springer-Verlag, 2024.

[56] Yao Tian, Ziyang Yue, Ruiyuan Zhang, Xi Zhao, Bolong Zheng, and Xiaofang Zhou. Approximate nearest neighbor search in high dimensional vector databases: Current research and future directions. *IEEE Data Eng. Bull.*, 46(3):39–54, 2023.

[57] Yao Tian, Xi Zhao, and Xiaofang Zhou. DB-LSH 2.0: Locality-sensitive hashing with query-based dynamic bucketing. *IEEE Trans. on Knowl. and Data Eng.*, 36(3):1000–1015, 2023.

[58] Godfried T. Toussaint. The relative neighbourhood graph of a finite planar set. *Pattern Recognition*, 12(4):261–268, 1980.

[59] Karthik V., Saim Khan, Somesh Singh, Harsha Vardhan Simhadri, and Jyothi Vedurada. BANG: Billion-scale approximate nearest neighbor search using a single GPU. https://arxiv.org/abs/2401.11324, 2024.

[60] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. Milvus: A purpose-built vector data management system. In *Proc. 2021 Intl. Conf. on Management of Data*, SIGMOD '21, pages 2614–2627. Association for Computing Machinery, 2021.

[61] Mengzhao Wang, Lingwei Lv, Xiaoliang Xu, Yuxiang Wang, Qiang Yue, and Jiongkang Ni. Navigable proximity graph-driven native hybrid queries with structured and unstructured constraints. https://arxiv.org/abs/2203.13601, 2022.

[62] Mengzhao Wang, Weizhi Xu, Xiaomeng Yi, Songlin Wu, Zhangyang Peng, Xiangyu Ke, Yunjun Gao, Xiaoliang Xu, Rentong Guo, and Charles Xie. Starling: An i/o-efficient disk-resident graph index framework for high-dimensional vector similarity search on data segment. *Proc. ACM Manag. Data*, 2(1), 2024.

[63] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *Proc. VLDB Endow.*, 14(11):1964–1978, 2021.

[64] Zeyu Wang, Peng Wang, Themis Palpanas, and Wei Wang. Graph- and tree-based indexes for high-dimensional vector similarity search: Analyses, comparisons, and future directions. *IEEE Data Eng. Bull.*, 47(3):3–21, 2023.

[65] Yuexuan Xu, Jianyang Gao, Yutong Gou, Cheng Long, and Christian S. Jensen. irangegraph: Improvising range-dedicated graphs for range-filtering nearest neighbor search. *Proc. ACM Manag. Data*, 2(6), December 2024.

[66] Artem Babenko Yandex and Victor Lempitsky. Efficient indexing of billion-scale datasets of deep descriptors. In *Proc. 2016 IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 2055–2063, 2016.

[67] Shuo Yang, Jiadong Xie, Yingfan Liu, Jeffrey Xu Yu, Xiyue Gao, Qianru Wang, Yanguo Peng, and Jiangtao Cui. Revisiting the index construction of proximity graph-based approximate nearest neighbor search, 2024.

[68] Yuanhang Yu, Dong Wen, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. GPU-accelerated proximity graph approximate nearest neighbor search and construction. In *Proc. 38th IEEE Intl. Conf. on Data Engineering*, pages 552–564, 2022.

[69] Weijie Zhao, Shulong Tan, and Ping Li. Song: Approximate nearest neighbor search on GPU. In *Proc. IEEE 36th Intl. Conf. on Data Engineering*, pages 1033–1044, 2020.

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