École Polytechnique Fédérale de Lausanne

Explore techniques to scale HNSW vector indexes with workload knowledge

by André Espírito Santo

Master Thesis

Approved by the Examining Committee:

Prof. Dr. Anastasia Ailamaki
Thesis Advisor

Ioannis Alagiannis and Martin Brugnara
External Expert

Antonio Boffa and Kyoung-Min Kim
Thesis Supervisor

EPFL IC IINFCOM DIAS
BC 222 (Bâtiment BC)
Station 14
CH-1015 Lausanne

September 11, 2025

Acknowledgments

I would like to start by thanking Prof. Anastasia Ailamaki for giving me the opportunity to work in association with the Data Intensive Application and Systems Lab for the last six months. I would like to thank Antonio Boffa and Kyoung-Min Kim for supporting me throughout my thesis and guiding me on this academic journey. I would like to thank Ioannis Alagiannis, Martin Brugnara and Vlad Haprian for introducing me to this problem, and mentoring me throughout the project.

Last but not least, I would like to thank my family, my girlfriend and my friends, for all of the support provided during this project. Without them, this would not have been possible.

Lausanne, September 11, 2025 André Espírito Santo

Abstract

AI agents and Large Language Models (LLMs) have become ubiquitous in the contemporary technological landscape. These agents depend on Vector Database Management Systems (VDBMSs) to efficiently store and retrieve large collections of high-dimensional vector data, necessitating sub-second query latency to enable real time interaction. To satisfy these demands, vector indexes have emerged as essential components within such systems.

Among these, graph-based vector indexes — specifically Hierarchical Navigable Small World (HNSW) [37] — are widely recognized as *state-of-the-art* [24, 37, 44], gathering considerable attention from both industry and academia. HNSW is an in-memory, multi-layered proximity graph that requires all vectors to be maintained in main memory to ensure efficient search operations. A critical parameter in HNSW is *ef_Search*, which governs the size of the dynamic candidate list maintained during query search, which is a *beam search* guided by the vector distance to the query vector. Increasing *ef_Search* enhances recall by expanding the search frontier but simultaneously incurs greater computational overhead and latency.

However, as dataset sizes grow, the memory requirements to store all vectors rapidly surpass the capacity of commodity hardware, rendering a purely in-memory index impractical. In practice, query workloads are often highly skewed and clustered [36, 42, 43], implying that only a small fraction of the graph is typically accessed to resolve queries. Building on this insight, we propose caching only a carefully selected subset of vectors in memory — those most frequently needed for query resolution — while offloading the remaining vectors to disk storage, thus significantly reducing memory usage.

Given a fixed *memory budget*, we introduce methods for selecting which nodes to retain in memory, guided by a small training sample used to model the incoming workload. These methods aim to minimize the number of disk I/O operations required to load uncached vectors and, more specifically, to maximize the proportion of queries that can be resolved entirely in-memory.

Among the proposed approaches, Heat Kernel PageRank (HKPR) demonstrates superior performance across various configurations. In most cases, it answers the greatest number of queries with minimal or no I/O operations compared to other methods - showing two to five times improvements - and achieving the highest average in-memory hit rate, indicating the largest number of *cache hits*. Notably, HKPR can deliver robust results with a very small training set relative to the incoming workload due to its generalization capability, delivering results up to tens of times better than other methods, which suffer from severe performance degradation as training set size decreases, a consequence of their limited generalization ability. Furthermore,

we examine the impact on recall when the search is confined solely to the subgraph of cached nodes, ignoring the *cache misses*. We demonstrate that when the cached nodes cover 95% or more of those accessed during search, the recall degradation from ignoring the remaining nodes is negligible, enabling those queries to be answered without any I/O operations and substantially improving latency. Depending on the dataset, queries with a substantially smaller fraction of nodes cached may also be answered entirely from memory with minimal recall impact.

Additionally, we compare our techniques with DiskANN [24], a well-known hybrid indexing method addressing HNSW's memory constraints. Although DiskANN requires significantly fewer full-precision vectors in memory by utilizing quantized vectors for graph navigation, it remains workload-agnostic. Consequently, our workload-aware methods, particularly HKPR, necessitate fewer disk fetches for the same *memory budget* and thereby incur fewer I/O operations, necessitating up to 5 times fewer vectors to be fetched from disk.

Acknowledging that disk I/O introduces substantial latency relative to a fully in-memory index, we further explore how to leverage workload knowledge to mitigate this effect. In the HNSW construction algorithm, nodes in upper layers are uniformly randomly selected from the layer below. We propose biasing this construction by overrepresenting vectors from frequently accessed regions of the *search space* ("hot" areas) while underrepresenting those from less frequently accessed regions ("cold" areas). This strategy seeks to reduce the number of distance computations during query execution by shortening the path from the search entry point to the top-ranked results. We conclude that this strategy significantly impacts performance for low *ef_Search* values, yielding a significant reduction in distance computations. However, for high-recall queries requiring high *ef_Search*, the impact is negligible. This is because most distance computations occur within the neighborhoods of the results rather than along the path from the entry point to the results themselves, due to increased graph exploration.

Contents

Chapter 1

Introduction

Recent advances in Machine Learning have led to the widespread adoption of AI agents and LLMs. These agents are highly dependent on the ability to store and retrieve high-dimensional vectors to address limitations in scalability, contextual relevance, real time information access, and generative reasoning capabilities. LLMs - and more generally, Machine Learning models - transform text, images, or other data types into vector *embeddings*, which are high-dimensional representations capturing the semantic content and context of the input data. Such embeddings are often stored for subsequent retrieval. A well-known example of this paradigm is Retrieval Augmented Generation (RAG). Within RAG, the LLM retrieves and utilizes stored embeddings that are deemed relevant to the current query, thereby enabling it to produce an informed response.

Systems designed to store and efficiently retrieve large collections of vectors are known as Vector Database Management Systems (VDBMSs). Since these systems are required to answer queries in real time, they must efficiently identify a small subset of relevant vectors—often in sub-second latency—from among, potentially, billions of vectors, each of thousands of dimensions. Consequently, **vector storage and retrieval** has become a cornerstone to modern large-scale information retrieval systems. Efficient handling of large-scale vector collections is critical for powering applications such as image search, recommendation engines, and natural language processing. This fundamental challenge is known as the **k-ANN** problem, where the objective is to retrieve the k approximate nearest neighbors of a given query vector with respect to a chosen distance metric (e.g., cosine similarity, Euclidean distance). Allowing for approximation (as opposed to exact k-NN) has enabled these algorithms to scale to collections comprising billions of high-dimensional vectors.

HNSW is a leading **in-memory** index structure for Approximate Nearest Neighbor (ANN) search [37]. It is widely implemented in VDBMS to index collections of vectors and to efficiently resolve k-ANN search queries. This index draws inspiration from the concepts of both *skip lists* [53] and Navigable Small World (NSW) [38] graphs. A *skip list* is a probabilistic data structure

that supports efficient $\mathcal{O}(\log n)$ search and insertion operations on a sorted array. It comprises multiple layers, where layer 0 corresponds to the full sorted array, and each subsequent layer contains a randomly selected subset of the elements from the previous layer. Therefore, upper layers contain fewer elements and allow the search process to efficiently skip large portions of the array. Once search converges to a particular element in the uppermost layer, it continues from that position in the next lower layer, providing a progressively finer-grained search until reaching layer 0, where the final result is obtained.

NSW graphs, on the other hand, are proximity graphs optimized for fast, Approximate Nearest Neighbor search. In these graphs, a *greedy* search is expected to traverse from any node to another in a logarithmic or polylogarithmic number of hops with respect to the total number of nodes.

HNSW combines these two concepts to construct a multi-layered proximity graph, where nodes correspond to vectors and edges connect similar vectors within the vector space. Layer 0 contains all nodes, while each layer above contains only a randomly selected subset of the nodes present in the layer below. HNSW is regarded as *state-of-the-art* in both academia and industry due to its low latency, high throughput, and scalability, rendering it highly effective for large-scale scenarios [24, 37, 44]. To answer a query, the algorithm initiates a *greedy* search, guided by a distance metric relative to the query vector, at the topmost layer of the HNSW index. At each step, it selects the closest node to the query vector search has already visited but not yet explored, and explores its neighbors. The algorithm maintains two priority queues, each storing at most j elements. In all layers but layer 0, $j = 1$, yielding a pure best-first search. However, at layer 0, j is set to *ef_Search*, a tunable parameter that presents a tradeoff between recall and latency. Thus, the search process effectively performs a *beam search* with beam width *ef_Search*, guided by distance to the query. For each neighboring node, if the queue has not yet reached j elements, or if the neighbor's distance to the query is smaller than the current maximum in the queue, it is added to both queues. The next node to visit is chosen as the closest element in the queue to the query, and the process is repeated. Search concludes when there are no more elements in one of the queues, though early termination strategies are also possible [37]. More details about the inner workings of HNSW are provided in section 2.4.

1.1 Motivation

Despite its status as *state-of-the-art*, HNSW also faces limitations, particularly as dataset size grows. Modern information retrieval systems have to store millions or even billions of vectors. **For datasets this size, storing all vectors and associated graph metadata in main memory often becomes impractical.** For instance, the commonly used SIFT1B benchmark contains 10^9 vectors, each with 128 dimensions. Assuming 4 bytes per dimension, the vectors alone require approximately $\frac{10^9 \cdot 128 \cdot 4}{1024^3} \approx 476.84$ GiB of storage, which exceeds the memory capacity of most commodity hardware. **This requirement also makes it difficult for a single instance hosting**

multiple such indexes for different datasets, even if modestly sized.

To address these challenges, various approaches have been proposed. Some methods alter the graph structure to facilitate serving vectors directly from disk while minimizing query latency and maximizing throughput. Examples include DiskANN, FreshDiskANN, and SPFresh [24, 58]. However, these methods necessitate that each query fetch data from disk, resulting in expensive I/O operations. Moreover, these approaches often employ vector *quantization*, where individual vectors are stored in a reduced, lossy format to save memory, at the cost of some loss in recall. This reduction in precision may not be acceptable for certain applications.

As an example, DiskANN relies on quantized vector representations in memory to guide the search, while the associated graph structure must still be fetched from disk at each search step. The last step of search involves re-ranking candidates using their full-precision vectors, which are stored on disk. Despite optimizations such as pre-fetching and *piggybacking*, which help reduce latency, disk access remains a considerable bottleneck. Furthermore, the memory required to store all quantized vectors can itself be substantial.

Importantly, all of the aforementioned approaches are *workload agnostic*: they do not exploit knowledge of real query workloads. As a consequence, these systems typically implement an "all-or-nothing" approach—either all vectors are kept on disk, or all reside in memory (the quantized vectors). While such strategies are convenient in that they require minimal workload assumptions, they overlook a crucial aspect.

In real-world industrial workloads, query access patterns are often highly skewed and clustered. This means that certain vectors are accessed far more frequently than others, resulting in clusters of activity [36, 42, 43]. Consequently, the majority of vectors may rarely, if ever, be retrieved. For this reason, we investigate how exploiting workload trends can enhance the HNSW index.

Notably, the concept of workload-sensitive optimization is hardly new in traditional database systems. *Buffer caches* have long been deployed to retain disk blocks likely to be required imminently, employing policies such as Least Recently Used (LRU) or Least Frequently Used (LFU) to manage what remains in memory and what is offloaded to disk. Recent research [32] has focused on adaptively managing buffer pools to prioritize frequently accessed ("hot") pages, balancing recency with storage costs, and often achieving significant performance gains in terms of number and pattern of SSD accesses for traditional database workloads.

1.2 Problem

This motivates us to explore solutions to reduce the index memory footprint by limiting the subset of vectors (or quantized vectors) to keep in memory, while the rest is offloaded to disk. Offloading vectors to disk presents a tradeoff between latency and memory footprint. Offloading

more vectors reduces the index memory footprint but also increases the likelihood of having to fetch vectors from disk during search, increasing latency. In order to try to alleviate the increase in latency caused by disk I/O operations, we need to analyze what impacts query latency in HNSW queries, and if there is any improvement possible based on workload knowledge (subsection 1.2.1). Equipped with this knowledge, we then tackle the problem of selecting the optimal subset of memory-resident full-precision (or quantized) vectors (subsection 1.2.2).

1.2.1 Latency

As an initial exploration to build a deeper understanding of the problem space, we investigate how workload information can be used to optimize entry point selection in the HNSW index. Within the standard HNSW construction algorithm, nodes in layer $l+1$ are selected uniformly at random from the nodes present in layer l. This strategy guarantees a uniform spatial distribution of entry points for each layer throughout the search space.

Nevertheless, under the assumption of a clustered query workload, alternative strategies may yield improved entry point selection. Because the upper layers of the index contain exponentially fewer nodes than layer 0, not all nodes in the base layer are positioned close to an entry point. Query latency depends on the number of distance computations performed. Hence, starting search closer to the query - and, therefore, closer to the result - generally reduces the number of hops during search, thereby reducing the number of distance computations and latency. Our aim, then, is to develop a method that, for a fixed number of upper-layer nodes, reduces the number of distance computations—and potentially enhances recall. The intuition underlying our approach is to overrepresent frequently accessed regions of the *search space* ("hot" areas) with a higher density of entry points, while sparsely accessed regions remain underrepresented ("cold" areas). Thus, queries concentrated in clustered workloads would be surrounded by multiple entry points, whereas areas with infrequent query activity would have correspondingly fewer entry points.

Several alternative strategies for entry point selection have previously been proposed. For instance, [44] demonstrates that NSW graphs often reveal the presence of *hub* nodes, suggesting these high-connectivity nodes could serve as entry points. Another study [45] proposes selecting as entry points nodes with low *Local Intrinsic Dimensionality*. These results collectively indicate that more effective entry point selection methods are attainable beyond uniform random sampling.

To evaluate our proposed intuition, we performed an optimal case study. For each query, we recorded its actual entry point in layer 0 within the HNSW index, along with its *top-K* results, layer-wise count of distance computations, and recall at $k = 1, 10, 100$. We then re-ran each query, but restricted the search to layer 0, forcibly starting search at the node corresponding to the top-1 returned result from the prior run. While this setup is strictly theoretical as it assumes *a priori* knowledge of the top-1, it enables us to estimate the practical impact that a *workload-aware*

entry point selection strategy might achieve.

Our experimental findings reveal that this strategy consistently reduces the number of distance computations, particularly at low values of *ef_Search*. For instance, we observe an approximate 20% reduction for *GIST* and 27% for *GloVe100* with *ef_Search* = 16. However, the benefit rapidly diminishes as *ef_Search* increases, becoming negligible for *ef_Search* = 512 across all datasets. This is because with high *ef_Search* values, a substantial portion of the graph is traversed regardless of the entry point, so the advantage of being nearer the query vanishes as the volume of distance computations increases. Conversely, at low *ef_Search*, the effect is most pronounced. Notably, the use of the top-1 node as the entry point produces only minimal changes in recall. Overall, recall differences remain small, with a maximum observed improvement of 5% at the lowest *ef_Search*, and tending towards zero otherwise.

Finally, we compared the performance of a flat NSW graph to that of the hierarchical HNSW structure. Our results mirror the earlier findings: while for small *ef_Search* the flat NSW sometimes incurs slightly more distance computations (although the reverse is true for some datasets), the gap rapidly closes as *ef_Search* increases. The root cause, again, is the insensitivity of wide beam searches to entry point proximity. As in the previous results, recall differences remained negligible.

1.2.2 Memory

As previously discussed, memory can become a significant bottleneck when dealing with large datasets. To address this issue, we propose methods for assigning a *cache priority* to each node in the graph. Given a particular method and a specified *memory budget*, nodes with the highest priority are retained in memory, while the remainder are offloaded to disk.

Our goal is to develop a strategy that, given some queries sampled from the clustered query workload and a fixed *memory budget*, selects a subset of vectors from the original dataset to be cached in memory (with the remainder offloaded to disk) in such a way that query latency for the given workload is minimized. In contrast to approaches such as DiskANN [24], which keep all quantized vectors in memory and all full-precision vectors on disk, our approach aims to selectively retain only those full-precision vectors that are likely to be needed for the anticipated workload, offloading the rest to disk. Not using quantized vectors keeps the recall unchanged, compared to the in-memory HNSW. Moreover, as either the dataset size or the dimensionality of each vector increases, storing all quantized vectors in memory becomes impractical due to the substantial storage requirements.

The parameter *memory budget* thus enables a trade-off between memory consumption and query latency: increasing *memory budget* allows more vectors to be cached in memory, thereby reducing the number of I/O operations required to answer a query, but at the cost of higher memory usage. Importantly, such strategy can be applied to scale existing quantization-based

solutions like DiskANN, since it can be applied to select a subset of quantized vectors to keep in memory.

In contrast to *buffer caches*, we propose a static cache. For high values of *ef_Search* — which are often required to achieve high recall — each query may access a substantial number of vectors when performing a search in the HNSW index. If a dynamic cache was employed (e.g., using LRU), a few outlier queries could entirely disrupt the cache by replacing the cached nodes associated with the main workload with nodes accessed only by these outliers. This would subsequently force a reload of the vectors most frequently required by the majority of queries. This topic is further discussed in section 6.1.

In our approach, a subset of sampled queries from the workload is used to determine which nodes are retained in memory. When a query accesses a vector not present in the cache, the vector is loaded, used, and then discarded. The cache contents are only updated during a repopulation phase, which is triggered by sampling new queries to decide which nodes should remain in memory.

We consider that detecting when a query workload is clustered (and thus when our approach is effective, as demonstrated in section 6.3) and identifying when the workload distribution shifts — necessitating the reassignment of vectors to memory by re-running the method — are both important but orthogonal challenges, which we leave as future work.

This work assumes that the graph structure is maintained in memory to efficiently compute cache priorities for nodes. For high-dimensional vector collections, the memory footprint of the graph structure is negligible compared to that of the vectors, which dominate storage costs. While the proposed methods could be adapted to store the graph structure on disk, this is left for future investigation.

In multi-layered graph-based indexes such as HNSW, the upper layers (both graph structure and vectors) are assumed to be kept in memory, as each layer contains an exponentially smaller subset of vectors than the layer below, constituting a minor portion of total memory usage. Some vectors present in upper layers may overlap with cached base layer vectors. Nevertheless, our methods can be used to selectively cache a subset of upper layer nodes in memory, offloading the remainder to disk.

Lastly, we assume that the index is constructed entirely in memory. There exist methods [18, 24, 74] for building HNSW indexes when the vectors cannot all reside in memory simultaneously. For instance, DiskANN proposes an approach for building the Vamana index, which can be adapted to HNSW. In distributed environments, another option is to construct the index in a distributed manner and subsequently serve the completed index from a single node.

As a baseline, we employ the Most Frequently Used (MFU) policy, because it is a well-established policy [15, 47, 52] and intuitive cache prioritization strategy in this problem's context. In clustered query workloads, certain regions ("hot" regions) experience frequent access, whereas

others ("cold" regions) are rarely or never accessed. Accordingly, our baseline method consists of caching those nodes most frequently accessed by a set of training queries representative of the target workload. In subsection 6.2.1, we highlight and empirically examine the limitations of this strategy: a single query may traverse a large number of nodes before converging, and even similar queries may visit distinct sets of nodes. Crucially, the set of nodes accessed by each query depends on the underlying graph structure, a factor not considered by the MFU policy. Given a set of training examples, the MFU approach cannot generalize to nodes that similar—but unseen—queries might visit. Consequently, if the *memory budget* allows caching more nodes than those accessed by the training queries, all nodes seen during training are retained while the remainder are selected at random, since no information is available for them. Thus, for the MFU policy to yield optimal results, the distribution of visits for each node within the entire workload must be closely reflected in the training queries, implying that a large and representative set of training queries is required.

To overcome these shortcomings, we explore alternative graph-based strategies, specifically Expand from the Visited Set (EVS), Personalized PageRank (PPR) and Heat Kernel PageRank (HKPR). The first is an improvement on top of the baseline. If the *memory budget* is smaller than the number of nodes visited by train queries, it employs the same strategy as the baseline. However, if this is not the case, instead of choosing the remaining nodes randomly like the baseline, it performs a Breadth-First Search (BFS) from the set of visited nodes, and the cache priority of such nodes is inversely proportional to the number of hops to a node visited by a query, ensuring nodes close to other nodes that were visited by train queries are also kept in memory. The advantages and shortcomings of this method are further analysed in subsection 6.2.2. The last two are random walk models that propagate the access frequencies of labeled nodes (i.e., those visited during training) throughout the graph according to its structure and node degree distribution. By simulating random walks, these methods allow generalization from the training queries to structurally similar, yet unseen, queries and account for the overfitting of train access frequencies to the training set itself. As a result, such methods better estimate the frequency with which each node would be accessed by other similar queries.

To empirically evaluate our approach, we devised clustered query workloads from well-known ANN benchmark datasets, namely *GloVe100*, *GIST*, and *SIFT*, with dataset sizes ranging from 1M to 50M and vector dimensionality ranging from 100 to 960. The rationale for not using the original query sets and the methodology for constructing clustered workloads are detailed in chapter 4. From the generated dataset, we split the queries into equal-sized training and test sets: the training set is used to assign cache priorities via the various methods, while the test set is used to evaluate the performance of these assignments. Our evaluation considers several metrics. Initially, we assess, across different workloads, datasets, *ef_Search*, *top-K*, and *memory budget* values, the average percentage of nodes from those accessed by a query that are included in the cached subset. In our experiments, *ef_Search* ranges from 32 to 512 and *memory budget* from 0.4% of the dataset to 90%. While HKPR consistently outperforms other methods, the overall differences—particularly when compared to the baseline—are relatively modest in this metric. However, this average can hide significant variations across individual queries. As

our primary objective is to maximize the number of queries that can be served entirely from memory, we further evaluate the fraction of queries for which over 99% of accessed nodes are in the cached subset. For reference, these are queries for which either all of the accessed nodes are cached or only a very small percentage (< 1%) is on disk. Here, HKPR demonstrates a decisive advantage: the baseline performs poorly due to its inability to generalize from training data and its ignorance of the graph structure. As a result, even with higher *memory budget* values, most vectors have a cache priority of zero, leaving their inclusion to chance. Because the baseline neglects the graph's connectivity, gaps may be introduced in the search path of HNSW, negatively impacting the fraction of queries that can be fully answered from memory. Notably, we observe that HKPR can accommodate far more queries with all accessed nodes cached in memory at a lower *memory budget* compared to both the baseline and EVS. In some datasets, EVS requires a *memory budget* value four times that of HKPR to serve a similar percentage of queries. Using this metric, the baseline is significantly outperformed: for the same *memory budget*, HKPR can successfully answer ten times more queries than the baseline across all datasets tested. This is due to HKPR and PPR generalization ability, which allows them to use the graph structure and degree distribution of nodes to smooth the visited frequencies recorded during execution of the training queries, which makes these priorities better reflect the visited frequencies of the entire workload.

Furthermore, we observed that the proportion of queries for which all accessed vectors are already present in memory is substantially smaller than the proportion of queries with at least 99% of the accessed nodes in memory, even in the case of HKPR. Since our primary objective is to maximize the number of queries answered without performing any I/O operations, we investigate the strategy of disregarding vectors that are not cached (and would therefore require loading from disk) and restricting the search to the subgraph composed exclusively of cached vectors and their corresponding edges.

For HNSW, if the entry point in layer 0 is not in memory, or if all of its neighbors are missing from memory — thus halting the search immediately — we instead initiate the search from an alternative entry point that is cached in memory and has at least one neighbor also cached. The results of this experiment are presented in subsection 6.3.7. Our findings indicate that — once the value of *memory budget* exceeds a certain threshold — the differences in recall become negligible, on average. This suggests that vectors stored on disk can, in such cases, be safely ignored.

Notably, across all datasets, queries for which at least 95% of the accessed nodes in the original search are in memory exhibit only a minimal and negligible drop in recall. This implies that these queries can, in principle, be answered entirely from memory, without any I/O.

Examining the distribution of recall loss across all queries, we find that HKPR is able to answer the majority of queries without any significant decrease in recall. Although the maximum drop in recall is larger for HKPR than for the baseline, this occurs in fewer than 10% of all queries. For the remaining queries, HKPR yields a recall drop that is equal to or less than that of the

baseline. It is plausible that the queries experiencing a larger recall drop with HKPR are outliers that do not conform to the general trend captured by HKPR but happen to benefit from a few randomly chosen vectors considered by the baseline, thereby achieving a smaller drop in recall.

Lastly, across all datasets, we demonstrated that - under the assumption that the training set is representative of the workload - PPR, and in particular HKPR, exhibit considerably lower sensitivity to the size of the training set when compared to the other methods. This implies that even with a relatively small number of training queries, these approaches are capable of generalizing effectively to the underlying workload, with the resulting performance closely matching that obtained using substantially larger training sets. In contrast, alternative methods do not exhibit this property: when trained with a small number of queries, their performance deteriorates significantly relative to scenarios where more queries are considered for training. For instance, with only 45 training queries on *SIFT*, or 450 queries on *SIFT10M*, HKPR already surpasses the performance of the baseline as well as that of EVS trained with 900 and 9000 queries, respectively, on the same test set, containing 2100 testing queries for *SIFT* and 21000 queries for *SIFT10M*. Consequently, PPR, and particularly HKPR, are especially well-suited for settings in which the training set is substantially smaller than the anticipated workload, provided that the training data remains representative of that workload.

Despite the notable results of HKPR — which allows a substantial proportion of queries to be answered from memory, even with a relatively limited *memory budget* — this method (and more generally, this approach) is not without its limitations. First, while the used datasets are widely recognized, the query workloads used here are synthetically generated, as explained in chapter 4. Real-world workloads may diverge from these synthetic patterns, in which case the reported performance of our approach may not hold. Second, the problem of efficiently detecting when the workload has shifted is not addressed in this study and constitutes a non-trivial challenge on its own. Finally, the impact of mismatched configurations between training queries and the actual workload is not assessed, and is likely to reduce its effectiveness. For example, if training queries use a small *ef_Search* (e.g., *ef_Search* = 16), the resulting access patterns will be more localized; if, subsequently, the actual workload uses a much larger *ef_Search* (e.g., *ef_Search* = 512), each query will access many more vectors and the access distribution will be considerably different from the one used to select cached nodes. A similar situation arises with filtered queries: if the subset of cached nodes is selected without considering filters, but subsequent queries are filtered, they may access nodes entirely outside the chosen subset - because, for example, all of those nodes are invalid according to the filter - limiting cache effectiveness. Additionally, methods like HKPR and PPR incur a bigger runtime and memory footprint than EVS.

While there are methods to approximate these scores that achieve better scalability as the graph size increases [9, 33, 34, 73], they may still become impractical for extremely large graphs. One approach is to aggregate certain nodes and construct a new graph derived from the original HNSW layer 0 graph, where each node represents an aggregation of original nodes, and edges correspond to the union of edges between these aggregations in the original graph. Another approach is to disregard portions of the graph. For instance, we could restrict the computation

to the subgraph containing all nodes accessed by training queries, along with any nodes within at most 10 hops from these. This would reduce graph size, thereby lowering both latency and memory requirements. We leave a detailed investigation of these two strategies to future work. Alternatively, one could employ EVS, which yields lower accuracy than HKPR but provides reduced memory consumption and improved latency.

1.3 Contributions

The main contributions of this thesis are as follows:

1. We investigate common ANN benchmark datasets and demonstrate that they do not inherently exhibit clustered workload patterns, as evidenced by comparisons between training and query sets using `t-SNE` visualizations.
2. We introduce a novel method that receives as input a query set and some desired workload properties, and generates a query workload. We demonstrate empirically its effectiveness in producing the desired workload properties.
3. We present a comprehensive analysis of how recall and the number of distance computations change when entry points are placed closer to the query vector.
4. We provide a comparison of recall and distance computation trade-offs between NSW and HNSW.
5. We propose several methods for assigning cache priority to each vector in an HNSW index. These methods are *workload-aware* and incorporate both the graph structure and degree distribution.
6. We perform extensive evaluations of these methods across diverse datasets, workload distributions, search configurations, and varying values of *memory budget*. In addition, we investigate the effect on search recall when nodes excluded from the cache by each method are omitted during search.
7. We benchmark our proposed methods against another hybrid indexing technique, *DiskANN*.

1.4 Thesis Structure

In chapter 2, we introduce the fundamental concepts and background knowledge necessary to understand this thesis, ensuring it is self-contained. In chapter 3, we discuss the core research problem and its significance. In chapter 4, we precisely define the target workload and its

desirable properties. We analyze why query sets from standard ANN benchmarks are inadequate for our purposes and present our method for extracting clustered query workloads. In chapter 5, we further refine the subproblem of entry point selection, review alternative strategies, present our intuition, and validate it through a best case study. In chapter 6, we focus on the main subproblem of memory pressure. We define our baseline, introduce new methods along with their advantages and limitations, and conduct a comparative analysis using multiple metrics and configurations. In chapter 7, we review existing approaches, highlighting how they differ from our proposals or fail to meet our objectives. Finally, in chapter 8 we summarize the key findings from both subproblems and outline potential directions for future research.

Chapter 2

Background

In this section, we provide an overview of the fundamental concepts required to understand and contextualize the contributions of this thesis.

2.1 Vector Database Management System (VDBMS)

A Vector Database Management System (VDBMS) is a specialized database management system designed to efficiently store, manage, process, and query high-dimensional vectors [50]. Such vectors are common in modern AI applications, including RAG, document retrieval, and e-commerce recommendation systems, among others. As a result, VDBMSs form the backbone of these applications and are therefore crucial components in contemporary data architectures.

The ability to manage large collections of high-dimensional vectors and retrieve them with low latency is vital for achieving scalability and serving millions of users concurrently. Moreover, production systems typically require both high-throughput reads and frequent concurrent writes, all while ensuring transactional consistency. Additionally, they must satisfy stringent latency requirements, often returning query results within sub-second time frames. Given these challenges, the design and implementation of VDBMSs demand sophisticated solutions and remain an active area of research in both academia and industry.

Traditional database systems are not suited for managing vector data for several reasons. First, vector queries are based on semantic similarity, requiring the identification of nearest neighbors to a given query vector (section 2.2). Second, vectors lack a natural ordering, rendering standard indexing schemes ineffective. Third, vectors are typically high-dimensional, leading to substantial storage and retrieval overhead. Lastly, many real-world workloads demand hybrid queries that combine nearest neighbor search with conventional filters (for databases supporting both standard and vector data types, such as the Oracle Database), further complicating query

processing and optimization.

2.2 Approximate Nearest Neighbor (ANN) Search

In a (c, k)-search query, the goal is to retrieve the k most similar vectors to a query vector q, subject to the condition that none of the returned vectors has a similarity score that is a factor of c worse than the best (non-zero) score. The similarity score is defined by a distance function $d(v_1, v_2)$, such as Euclidean or Cosine distance.

When $c = 0$, the task is called a k-nearest neighbor (k-NN) query search, where the goal is to exactly retrieve the k closest vectors to q [50]. The only way to guarantee an exact solution is to compute $d(q, v)$ for every v in the collection, making the computational cost scale linearly with the dataset size. This approach is impractical for modern vector collections, which may contain up to billions of vectors in a VDBMS.

To address this, many systems employ $c > 1$, allowing for approximate solutions that trade off a small amount of accuracy for a significant gain in efficiency. Such queries are referred to as k-ANN queries: the set of k returned vectors is not necessarily the true set of k nearest neighbors, but lies within a bounded approximation.

For these approximate queries, various specialized vector indexes (section 2.3) have been developed. These indexes enable efficient similarity search at scale, significantly improving performance over the exhaustive k-NN approach.

k-ANN k-NN

Figure 2.1: k-ANN vs k-NN search

2.3 Vector Indexes

Vector indexes are specialized data structures designed to enable efficient search over high-dimensional vector data, thereby supporting practical and scalable ANN search. These indexes address challenges inherent to high-dimensional data, such as data sparsity and the so-called curse of dimensionality [50, 62]. In addition, the large size of individual vectors makes access and distance computations costly.

The various types of vector indexes offer different trade-offs among search accuracy, query latency, memory consumption, index construction time, and support for dynamic updates [11]. When deployed in a real-world database system, these indexes must further support transactional workloads and meet the ACID properties.

Broadly, vector indexes can be categorized into four groups: **graph-based approaches** [16, 24, 37, 60], **hash-based approaches** [5, 7, 23, 67], **quantization-based approaches** [1, 2, 63, 72] and **partition-based approaches** [36, 59].

2.3.1 Graph-based approaches

Graph-based approaches construct a proximity graph in which data points (i.e., vectors) are nodes, and edges connect pairs of nodes representing similar vectors under a chosen distance metric. Search queries are executed as graph traversals, where the algorithm navigates from a starting node toward approximate nearest neighbors using best-first or other greedy search strategies. These methods have demonstrated superior performance in scenarios demanding high recall and low latency [70], but typically do not provide formal accuracy guarantees (i.e., the value of c in a (c, k)-query is not known).

2.3.2 Hash-based approaches

Hash-based approaches follow a fundamentally different paradigm. Locality-Sensitive Hashing (LSH) — the most prominent example — ensures that similar vectors are likely to be mapped to the same hash bucket. Such methods offer formal guarantees on approximation quality (i.e., c is known) and sub-linear query times [64, 67], making them attractive where approximate answers with low latency are acceptable. However, they often require careful parameter tuning and may not achieve the same accuracy as state-of-the-art graph-based methods.

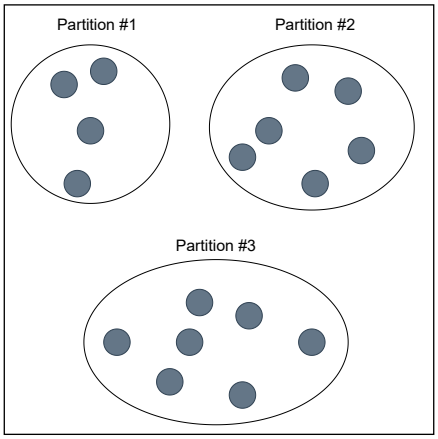


Figure 2.2: Search space partitioning employed by Inverted File Index (IVF)

2.3.3 Quantization-based approaches

Quantization-based approaches aim to minimize memory usage while maintaining search accuracy. Prominent techniques include Product Quantization (PQ) and Scalar Quantization (SQ), which compress high-dimensional vectors to enable efficient storage and retrieval. These quantization methods are frequently combined with graph or hash based indexes to form hybrid schemes that combine the strengths of multiple strategies [63, 72].

2.3.4 Partition-based approaches

Partition-based approaches subdivide the search space in partitions, in order to restrict search to the most promising partitions. This approach can also be combined with any of the aforementioned approaches. One well-known example is Inverted File Index (IVF) [36, 59], which divides the vector space into K partitions. This index is depicted in Figure 2.2. K is typically set to the square root of the dataset size, though it can also be user supplied. For each partition, a partition centroid is calculated, and the vectors are assigned to the partition with the nearest centroid. During search, all of the vectors belonging to partitions of the closest i centroids to the query are fully scanned to identify the closest vectors to the query. i controls the tradeoff between recall and latency. A bigger i induces more vector distance computations, increasing latency and recall, while a lower i induces less distance computations, reducing latency and recall.

2.4 Hierarchical Navigable Small World (HNSW) Index

HNSW [37] is one of the most prominent graph-based vector indexes, widely adopted across both academia and industry.

The central insight underlying HNSW is the use of NSW graphs [38], which allow greedy traversal algorithms to scale logarithmically or poly-logarithmically with the dataset size. A greedy graph traversal is a particular type of search algorithm that makes locally optimal choices at each step in the hope of finding a global optimum with respect to a metric. Therefore, in the context of vector search, NSW graphs are graphs where a *greedy* search can traverse from any node to another in $\mathcal{O}(\log^k n)$ hops, where n is the total number of nodes.

NSW networks are characterized by high clustering coefficients and small diameters, properties which give rise to a mixture of short and long edges and enable efficient navigation [37, 39]. The HNSW construction aims to approximate a Delaunay graph, which — if it could be efficiently constructed — would guarantee that greedy traversal always finds the true nearest neighbor. However, Delaunay graphs are difficult to construct in high dimensions without extensive prior knowledge, and thus serve primarily as a golden standard.

HNSW employs a hierarchical structure inspired by *skip lists* [53], resulting in a multi-layered, directed proximity graph with n layers. The nodes in each layer represent vectors, and edges connect similar vectors based on a chosen distance metric (such as Euclidean, Cosine, or Manhattan distance). The bottom layer contains all vectors. For each data point, the maximum layer on which it appears is randomly determined according to an exponentially decaying probability. This design ensures that upper layers are smaller and sparser, supporting rapid traversal across the vector space (the *zoom-out* phase), while lower layers are denser, supporting precise local search near the query (the *zoom-in* phase).

The search algorithm in HNSW is a best-first beam search, maintaining the best *ef_Search* candidates at each iteration, as illustrated in Figure 2.3. The algorithm starts at the top layer and descends layer by layer, searching for the closest vector to the query. For each layer, two sets are used: a candidate set C and a nearest neighbor set W, both of which hold up to *ef_Search* vectors. Both sets are initialized with the entry point for the current layer, which is the top-1 result from the layer above. At every iteration, the closest candidate to the query is popped from C and compared to the farthest neighbor in W. If the latter is smaller than the former, it means that all candidates in C are worse than the current worst element in W and no new element will ever be inserted in W, so search converges and returns the (up to) k closest elements in W. Search also naturally stops when $|C| = 0$. *top-K* can never be bigger than *ef_Search* because they are the k closest vectors to the query vector in W, where $|W| \leq$ *ef_Search*. The same applies for *M* and *ef_Construction* during construction, as *ef_Construction* and *ef_Search* serve the same role.

If search does not converge, all unvisited neighbors of the current candidate are explored. Each neighbor is added to W and C either if $|W| <$ *ef_Search* or if its distance to the query vector in the *vector space* is smaller than the worst element in W for the same metric, in which case the latter is popped from the queue. This process repeats until convergence. On all layers except the bottom, *ef_Search* is set to 1, in which case search only keeps the best seen element in W, then used as an entry point for the layer below when search converges. Only layer 0 uses a larger value. If building the index, *ef_Search* is referred to as *ef_Construction*.

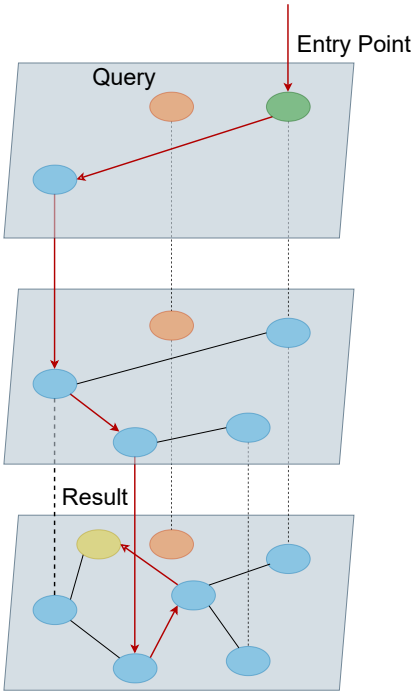


Figure 2.3: Hierarchical Navigable Small World (HNSW) index

Insertion begins by determining the maximum layer for the new node, then searching each eligible layer to identify nearest neighbors. The parameter M specifies the maximum number of neighbors per node (in layer 0 this is typically $2 \cdot M$). For each insertion, an initial selection of up to *ef_Construction* candidates is pruned to M using a heuristic. This heuristic ensures that - given a large enough *ef_Construction*- the selected candidates build a Relative Neighborhood Graph (RNG) as a subgraph, a minimal subgraph of Delaunay graph deducible by using only the distances between nodes [37]. These graphs state that two nodes p and q are connected *iff* no other point r exists such that $max(||p-r||, ||q-r||) < ||p-q||$ [37, 66]. Due to this heuristic, each layer has short and long edges within the graph, allowing for poly-logarithmic traversal characteristic to NSW.

Both M and *ef_Construction* are parameters used during construction. *ef_Construction* determines the number of candidate neighbors considered, so the higher the *ef_Construction*, the better the graph approaches the structure of an Relative Neighborhood Graph (RNG) at the cost of longer build times. M governs graph density, balancing memory usage, search latency, and recall.

During search, *ef_Search* controls the beam width: larger values provide higher recall and query latency by exploring more candidates at each step.

2.5 Buffer Cache

In database systems, a *buffer cache* refers to a dedicated region of main memory used to store recently accessed data blocks or pages from disk. The primary objective of the buffer cache is to minimize the number of I/O operations by retaining frequently accessed data in memory, thereby enabling much faster subsequent reads. Whenever the system receives a data access request, it first checks the buffer cache: if the data is present (*cache hit*), the request is served immediately from memory. If the data is absent (*cache miss*), the system must perform a disk I/O to load the required block. Typically, the newly loaded block is retained in memory, and an existing block is evicted from the cache according to a specified cache replacement policy.

Common cache replacement policies include:

- Least Recently Used (LRU): The block that has not been accessed for the longest period is evicted from the cache.
- Least Frequently Used (LFU): The block with the lowest access frequency is evicted.

The efficiency of these policies depends on factors such as access patterns and cache size. Neither policy is universally superior. Beyond replacement policies, there also exist *cache priority policies*. In scenarios where only a subset of candidate elements can be held in the cache, these policies assign a priority to each candidate, guiding the selection of which items to retain. Contrary to cache replacement policies, a higher cache priority means that the underlying data should stay in memory, as it is highly likely to be accessed. Two notable examples are:

- Most Recently Used (MRU): The caching priority is proportional to the recency of use; elements accessed most recently have higher priority.
- Most Frequently Used (MFU): The caching priority is proportional to usage frequency within a given timeframe; elements accessed most frequently have higher priority.

In chapter 6, we focus specifically on caching priority policies and examine their application in the context of VDBMS indexing structures.

2.6 Query Workload

Within the context of vector search, a *query workload* denotes the collection of vector similarity queries that a system is expected to process, along with any inherent characteristics or patterns in those queries. It is important to note, however, that no universally accepted definition of a query workload exists.

In this thesis, we define a query workload in terms of a query distribution. Specifically, this can be conceptualized as a probability distribution over $\mathbb{R}^d$, where d is the dimensionality of the query vectors. Concrete query vectors are then instantiated by sampling from this distribution. For the purposes of this work, we do not consider either the order or the arrival times of queries in the system.

This formulation enables the representation of any conceivable spatial distribution of queries in the vector space. However, our primary interest lies in **clustered** query workloads, characterized by a probability distribution that exhibits increased density in certain regions of the search space (i.e., queries are more likely to originate from those regions).

An illustrative example is the multivariate Gaussian distribution $\mathcal{N}_d(\mu, \sigma)$, centered at $\mu \in \mathbb{R}^d$, where σ determines the spread of the cluster in each dimension. In practice, the underlying probability distribution is seldom known explicitly and is instead inferred from observed samples of queries.

Chapter 3

Problem Statement

In this chapter, we provide a comprehensive overview of the central problem addressed in this thesis. We highlight the main limitations of the HNSW index and decompose the problem into two distinct subproblems, each of which is examined and refined in detail within the corresponding sections.

We also discuss the significance of this problem and how its resolution can benefit current implementations of HNSW. Finally, we introduce a set of common notations to be used throughout the thesis.

In this thesis, we explore techniques to scale HNSW vector indexes by exploiting workload knowledge. In particular, we explore:

1. how to reduce latency, using workload knowledge to smartly choose nodes present in the upper layers, and what is its impact.
2. how to reduce the memory footprint to further scale the HNSW index and/or allow more concurrent indexes to co-exist in the same computing instance, for a fixed *memory budget*.

In particular, we are interested in workloads which present trends - *clustered workloads* - further discussed in chapter 4. As an example, one could consider the embeddings related to trending videos on a streaming platform, or embeddings from products sold on an *e-commerce* platform, whose trends change over time and are, sometimes, seasonal. Since common ANN benchmarks do not include such workloads, an additional challenge is how to derive the desired workload from the pre-existing *query sets*, included in these benchmarks, also discussed in chapter 4.

3.1 Latency: Improving entry point selection

As described in section 2.4, during the construction of an HNSW index, nodes promoted to higher layers are selected at random, in accordance with the original algorithm [37]. This strategy ensures a uniform distribution of entry points across the graph, enabling efficient access to most nodes in layer 0 within a small number of hops. However, due to the exponential reduction in the number of nodes in higher layers compared to layer 0, an entry point may still be significantly distant from the query vector, both in terms of vector space distance and graph traversal hops. A straightforward solution is to increase the number of vectors present in the upper layers; nevertheless, this comes at the expense of greater index size, increased build time, and higher query latency.

Alternatively, rather than merely increasing the number of upper-layer nodes, more principled selection methods exist. Different approaches to solve this problem are described in chapter 7.

Importantly, these approaches are inherently *workload agnostic*. In other words, they assign equal importance to every region of the *search space*, irrespective of the actual query distribution. Therefore, given a fixed number of upper-layer nodes, such strategies cannot fully capitalize on the properties of a specific query workload.

In practical industrial scenarios, however, query workloads are rarely uniformly distributed. Instead, they tend to exhibit clustering within specific regions of the search space [36, 42, 43]. A more detailed definition of such clustered workloads is provided in chapter 4. When serving these workloads using an HNSW index, nodes are also not uniformly visited throughout the graph, with certain nodes being accessed substantially more frequently than others. As a concrete example, consider a large e-commerce platform with millions of different products, employing a semantic search system powered by a VDBMS. During the summer season, summer equipment (e.g. swim shorts, beach towels, etc...) would be significantly more queried than winter products (e.g. winter jackets, snow skis, etc...), resulting in vectors associated with summer items being accessed far more often.

Building upon this observation, we aim to develop novel methods that specifically address this shortcoming of HNSW. To mitigate the limitations related to entry point selection, one could, for a fixed number of vectors in the upper layers, concentrate these vectors in regions of the search space that correspond to popular products, such as summer equipment, while allocating fewer entry points in less frequently queried regions. This concept is explored in detail in chapter 5.

Reducing the number of distance computations is crucial for scaling the HNSW index. Such improvements can reduce latency and, potentially, enhance recall. If recall is improved relative to random entry point selection — under identical build and search configurations — then for a fixed recall target, it may become possible to use a lower *ef_Search* during search and/or

construct the index with lower values of *ef_Construction* and M. This would translate into shorter index build times for equivalent output quality. Additionally, since - for the same workload - the percentage of the graph visited by a certain workload decreases as the graph size increases (shown in subsection 6.3.4), such strategies could enable upper layers to contain fewer vectors, reducing the memory requirements of the index for the same result.

Summing up, we want to answer 3 research questions:

LATENCY_Q1 What is the maximum improvement in reduction of distance computations and recall?

LATENCY_Q2 How to choose the nodes to promote to the upper layers?

LATENCY_Q3 What is the impact of this strategy on recall and distance computations?

3.2 Memory: Reducing the memory footprint

The HNSW index is also associated with substantial memory requirements. The standard implementation assumes that all vectors are resident in memory to permit efficient computation of distances between a query and dataset points during graph traversal.

This assumption quickly becomes problematic for large datasets. For example, in the case of the *SIFT* 1B dataset, one must store 10^9 vectors, each with 128 dimensions. Assuming 4 bytes per dimension, this results in $128 \cdot 4 \cdot 10^9 = 512 \cdot 10^9$ Bytes ≈ 477 GiB of memory usage—excluding graph metadata [25]. If each graph layer uses adjacency lists, with each entry occupying 4 bytes, then in the worst case, layer 0 alone would require $4 \cdot 2 \cdot M \cdot 10^9 \approx 7.45 \cdot M$ GiB for graph data. In practical implementations, additional overhead arises from ensuring *ACID* properties and supporting hybrid queries (e.g., filtering on scalar attributes alongside index search), further increasing memory demands. This quickly renders HNSW impractical for very large datasets, or in scenarios where multiple indexes must be maintained concurrently on a single machine. Naively writing and reading vectors to and from disk is not viable, as this results in unacceptable query latencies — disk accesses are orders of magnitude slower than accessing main memory.

There are existent methods to address this limitation. They are discussed in chapter 7, as well as how they differ from our approach.

Despite their innovations, these methods remain inherently *workload agnostic*. Most employ quantized vectors, trading lower recall for improved latency, and all require multiple disk I/Os per search. Compared to the traditional HNSW, such techniques typically experience either reduced recall or increased latency, given their reliance on quantized vector search and/or the necessity of loading vectors or graph components from disk to re-rank results.

To address this memory constraint, we propose prioritizing the retention of vectors corresponding to frequently accessed regions — such as summer equipment — in memory, while offloading less relevant vectors to disk storage. Intuitively, this strategy ensures that the vast majority of queries can be satisfied entirely from memory, as they predominantly access vectors that are retained in main memory. Our approach to this problem is detailed in chapter 6.

By leveraging workload information in this manner, it becomes possible to reduce the memory footprint of the index. Such improvements would allow a single compute instance to support a larger number of indexes or accommodate larger datasets within the same memory budget.

Addressing this limitation is of considerable significance. Firstly, reducing the memory requirements of the HNSW index would enable large vector collections to be stored and retrieved on commodity hardware. This, in turn, would lower the costs associated with building and maintaining large-scale information retrieval systems, thereby broadening access and facilitating wider deployment. Secondly, the ability to efficiently and dynamically adjust the memory footprint of an index is particularly vital in *cloud computing* environments, where memory resources are allocated and shared among multiple processes and workloads. Such environments often experience episodes of memory contention or stress. Thus, minimizing the memory consumption of these indexes directly contributes to system stability and resource efficiency.

Summing up, we want to answer 3 research questions:

MEMORY_Q1 How to choose the nodes to keep in memory?

MEMORY_Q2 What is the impact of different search and *memory budget* configurations on the effectiveness of this strategy?

MEMORY_Q3 How many queries can we answer completely from memory?

3.3 Notation

In this section, we formalize the notation used throughout this thesis.

Let:

- $\mathcal{V}$ denote the set of all nodes in the graph, and $\mathcal{E}$ the set of edges. (This notation is also used interchangeably to represent the set of all vectors.)
- $\mathcal{Q}$ denote a multivariate probability distribution formally representing the clustered query workload.
- $q \sim \mathcal{Q}$ represent a query sampled from this distribution.

- A *visited* vector (or node) refer to any vector for which a distance computation is performed in layer 0 of the HNSW index.
- A vector (or node) in the search path refer to any vector that is extracted from queue C during search (see section 2.4).
- $visited(q)$ denote the set of all nodes visited by a query q in the graph.
- $topk(q)$ denote the k results retrieved by the index for a query q, for a given value of k.
- *base dataset* refer to the collection of vectors indexed by the HNSW index.
- *query dataset* refer to a collection of vectors used to query the index on the *base dataset*.

Throughout this thesis, the terms vector and node are used interchangeably: when discussing distance computations, vector refers to the underlying vector representation of each node, and when referring to structural aspects of the graph, vector and node are used as synonyms, since each node corresponds to a unique vector.

While additional notation may be introduced in subsequent chapters for context-specific discussions, the conventions established here remain valid throughout the entirety of the thesis.

Chapter 4

Query Workload and Dataset Exploration

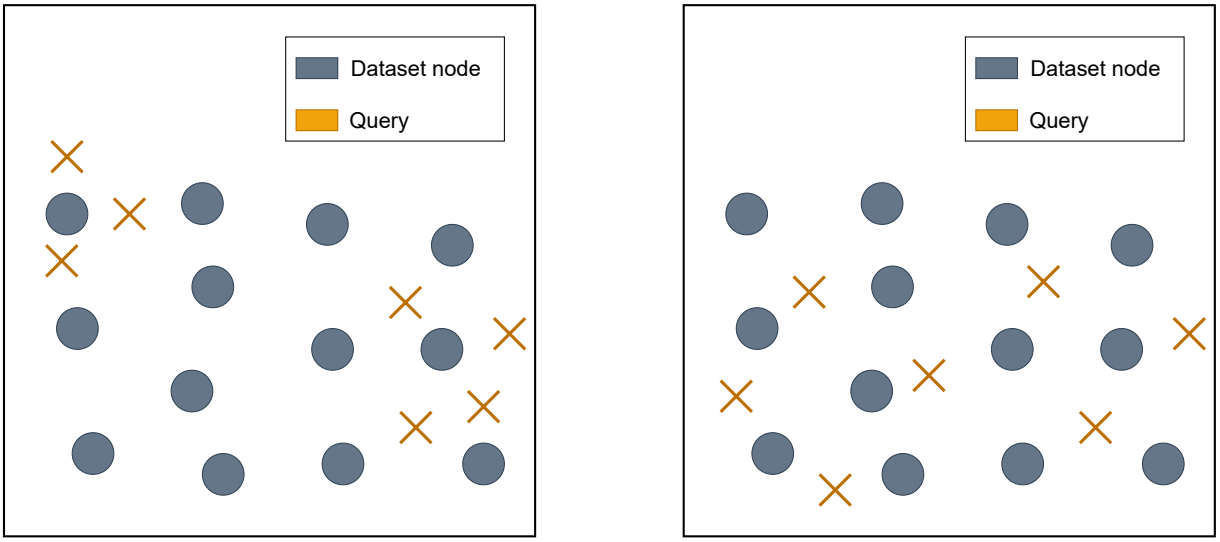
In this section, we define the assumptions underlying our query workload and provide justification for these choices. We examine the key characteristics of workloads of interest and evaluate commonly used benchmark datasets, highlighting the extent to which they exhibit, or fail to exhibit, these characteristics. Finally, we survey various techniques for generating such workloads from existing benchmarks and justify our choice of technique.

4.1 Clustered Query Workload

A *clustered query workload* is a query workload in which query vectors are organized into clusters within the vector space. Let the dimensionality of the vectors be denoted by d. We define the *search space* as the hyperrectangle in $\mathbb{R}^d$ bounded by the dimension-wise minimum and maximum values across all data points in the dataset. In a clustered workload, the distribution of data points (i.e., queries) throughout the *search space* is non-uniform. Instead, some regions have a higher density of data points relative to their surroundings. This distinction is illustrated in Figure 4.1 for the case $d = 2$.

4.1.1 Desired Characteristics of the Workload

The central motivation for assuming a clustered query workload is the ability to detect and exploit access patterns among the vectors. Such patterns permit algorithms to identify vectors that are infrequently accessed and are, therefore, eligible to be offloaded to disk with negligible impact on the performance of subsequent queries under the same workload. While this is an



Clustered Query Workload

Uniform Query Workload

Figure 4.1: Clustered vs Uniform Query Workload

assumption we make, it is common for real-life industrial query workloads to present these characteristics [36, 42, 43], as mentioned before.

For each query, multiple vectors are visited, as each step may require comparing the query vector with all neighboring nodes (determined by the parameter M). If the workload lacks clustering, or if the workload consists of an excessive number of clusters (or, equivalently, clusters of very large diameter), a substantial portion of the dataset will inevitably be accessed. Consequently, few vectors can be offloaded to disk without incurring significant additional latency, as many queries will require retrieving vectors from secondary storage, thereby substantially impacting overall query performance.

In other words, it is crucial to consider not only the distribution of queries but also the number of unique vectors visited by a given workload. For a workload to be appropriate for our experimental purposes, it should not result in visits to the entire set of vectors, for example. The number of unique vectors accessed is influenced by parameters such as M, *ef_Search*, and *ef_Construction*.

4.2 ANN Datasets: Generating Clustered Query Workloads

Several datasets are widely used to evaluate the performance of ANN search algorithms. In this work, we rely on *SIFT* (with variations containing 1M, 10M, and 50M vectors), *GloVe100* (1183514 vectors), and *GIST* (1M vectors) [25, 48, 51]. The *SIFT* and *GIST* datasets use *Euclidean* distance, whereas *GloVe100* uses the *Cosine* distance. *SIFT* has 128 dimensions, while *GIST* has

960 dimensions. *GloVe100* has 100 dimensions. All datasets are stored with the *float32* data type. As so, the vectors alone occupy $\approx 0.47\,GiB$ for *SIFT* ($\approx 4.77\,GiB$ for *SIFT10M* and $\approx 23.84\,GiB$ for *SIFT50M*), $\approx 0.44\,GiB$ for *GloVe100* and $\approx 3.58\,GiB$ for *GIST*. When hosting an HNSW index completely in memory, the additional storage cost of graph structure and all other possible metadata (such as metadata related to ensuring *ACID* properties) is to be added to storage cost of the vectors.

Each dataset provides a *base dataset* and a *query dataset*. To assess whether the *query dataset* exhibits the desired characteristics for this research, we utilize `t-SNE` [35], a well-established dimensionality reduction algorithm, to project the high-dimensional vectors into two dimensions while preserving relative distances. As a sidenote worth mentioning, we also used `UMAP` [41] as a dimensionality reduction algorithm, but as the results were similar, we only present `t-SNE`.

We then visualize both the queries and base vectors in scatter plots. This allows us to observe the distribution of query vectors across the *search space* and to identify whether natural clusters of activity arise.

Furthermore, we instrument the index to record how frequently each vector in layer 0 is visited across all queries. This offers additional insight into whether the *query dataset* satisfies the requirements established in subsection 4.1.1

4.2.1 Problem Statement

As we assume a clustered query workload, we need to verify our workload is indeed clustered. As shown in subsection 4.2.3, the query sets pertaining to these benchmarks do not exhibit such characteristic. Hence, we explore strategies to devise a clustered query workload from the pre-existing query sets.

4.2.2 Analysis Setup

All experiments in this section are conducted with *ef_Search* = 256, *top-K* = 10, and an HNSW index built using *ef_Construction* = 300 and *M* = 32. The base and query sets are sourced from `http://corpus-texmex.irisa.fr` and `https://nlp.stanford.edu/projects/glove/`.

4.2.3 Dataset Analysis

As illustrated in Figure 4.2, the *query datasets* generally cover most of the underlying dataset (with the exception of *GIST*, which includes only 1000 queries in comparison to 10,000 for the other datasets), and consequently do not demonstrate concentrated regions of high or low activity. Thus, we cannot directly utilize the entirety of the provided *query datasets* for our intended

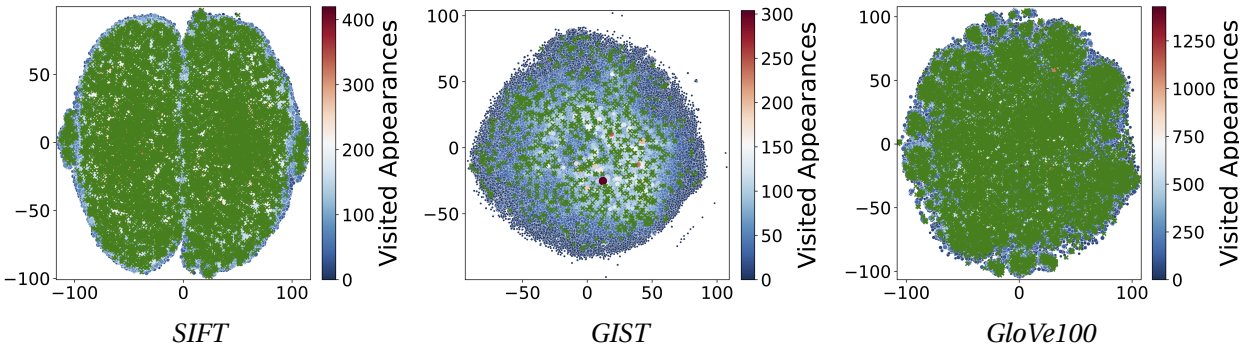


Figure 4.2: Dataset scatter plot. Queries are the green crosses while circles are data points. The size of the circle is correlated with the node in-degree in HNSW layer 0 and the color with the number of times the node was visited during search

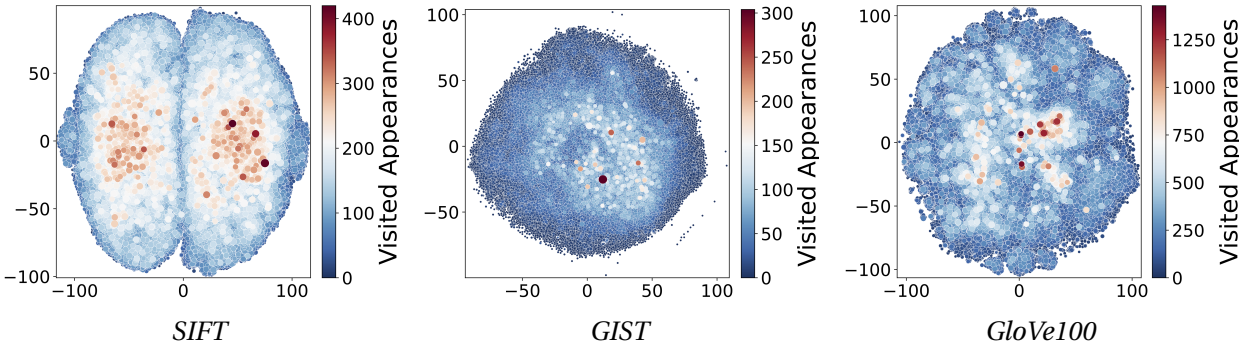


Figure 4.3: Dataset scatter plot. Circles are data points. The size of the circle is correlated with the node in-degree in HNSW layer 0 and the color with the number of times the node was visited during search

workload characterization. This is, however, intentional: these datasets are specifically designed to comprehensively cover the dataset, as queries are randomly selected and aim to stress-test all segments of the search space for a robust evaluation of ANN search algorithms. Hence, they are not designed to represent realistic workloads.

To better analyze access patterns, we remove individual queries in Figure 4.3. For both *GIST* and *GloVe100*, accesses are seen to be nearly uniformly distributed, with a few nodes—typically those with high in-degree (*hubs*) or designated entry points—appearing more frequently. This conclusion is also backed by previous reasearch [44]. For *SIFT*, two clusters of activity are noticeable. It is important to note that projecting hundreds of dimensions to two via `t-SNE` is inherently lossy, so while these plots offer valuable insights into access patterns, they too have their limitations.Most importantly, across all datasets, the vast majority of vectors are visited at least once, implying that nearly the entire dataset must remain in memory for efficient retrieval.

To construct a workload exhibiting the desired properties, we sample a subset of queries from the provided query sets. Several strategies can be considered for this selection.

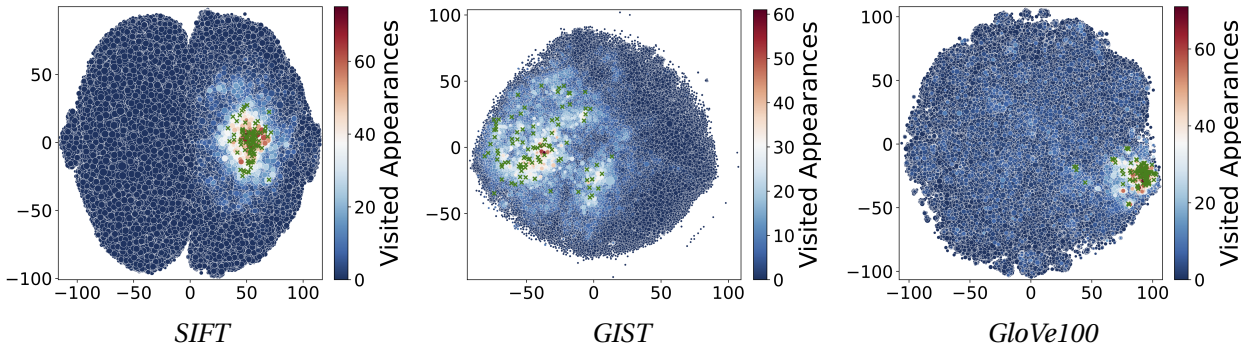
4.2.4 Workload Generation: Clustering queries

The most straightforward approach is to apply a clustering algorithm and pick queries from one or more clusters. Clustering algorithms can be divided into those requiring the number of clusters as a parameter (e.g., KMeans [27]) and those which do not. Using the former introduces the challenge of selecting the appropriate number of clusters. Various heuristics exist for choosing the appropriate number of clusters, which try to optimize different metrics of clustering quality: the Elbow method, Silhouette coefficient, Gap statistics, among others [4, 21, 55, 65]. However, this would add another experimental variable. Algorithms in the latter group are typically computationally intensive and provide less direct control over cluster diameter and, consequently, the resulting access patterns.

4.2.5 Workload Generation: Choosing the nearest neighbors of seed queries

Let $\mathbf{C}$ denote the desired number of clusters. We randomly select one seed query from the query set, and the remaining $\mathbf{C}-1$ seeds are chosen to maximize pairwise distances. Specifically, if $\mathbf{C}=2$, the second seed is selected as the vector furthest from the first. For $\mathbf{C}>2$, we employ the `Farthest Point Sampling` algorithm [13]. This way, each cluster has exactly 1 seed query. For each seed, we then use k-NN or k-ANN to retrieve the $\mathbf{L}$ closest queries according to the dataset's distance metric.

The parameter $\mathbf{L}$ controls the diameter of each cluster within the query set: geometrically, this corresponds to selecting all vectors within a hypersphere centered at the seed query, with a radius equal to the distance to its $\mathbf{L}^{th}$ neighbor. Increasing $\mathbf{L}$ broadens the radius, resulting in



SIFT *GIST* *GloVe100*

Figure 4.4: Dataset scatter plot. Queries are the green crosses while circles are data points. The size of the circle is correlated with the node in-degree in HNSW layer 0 and the color with the number of times the node was visited during search

wider clusters and influencing the number of nodes accessed in layer 0 of the HNSW index.

Figure 4.4 highlights the resulting access heatmap for $\mathbf{L} = 100$ and $\mathbf{C} = 1$, using the same values for *ef_Search*, *ef_Construction*, *top-K*, and *M* as previously. Notably, the selected queries are now localized within a specific region of the graph. Given the absence of a uniform query distribution, access patterns are also clearly clustered, and a portion of vectors remain entirely unvisited. These infrequently or never-accessed vectors are prime candidates for offloading to disk, as their absence from main memory will have minimal impact on the latency for the current workload.

Finally, we observe that queries in *GIST* are more spread out than those in *GloVe100* or *SIFT*. This is attributed to the smaller size of the *GIST query dataset*. Consequently, selecting $\mathbf{L} = 100$ nearest neighbors forms a larger hypersphere around each seed, causing individual queries to visit a broader set of vectors and resulting in less overlap in visited vectors among queries.

4.2.6 Conclusion

Choosing the nearest neighbors of C seed queries provides an easy way to control both the number of clusters as well as the cluster diameter. Without adding a new experimental variable, it provides results good enough to devise the type of *clustered workload* we intend to target.

Because of the aforementioned reasons, we decide to use this algorithm to devise a workload from the dataset *query sets*.

Chapter 5

Latency: Improving entry point selection

In this chapter, we investigate our first sub-problem, which concerns search latency. We begin by refining the problem statement, building on the broader introduction provided in chapter 3. Subsequently, we analyze how incorporating workload knowledge can guide the selection of entry points and further develop this intuition. Additionally, we present and interpret the results of a optimal case study that shows the maximum practical impact of these strategies on real-world searches within an HNSW index. Because these results were unexpected, we consider the opposite direction, and investigate the difference in distance computations and recall for a NSW index compared to an HNSW index.

5.1 Problem Statement

As outlined in section 2.4 and chapter 3, the maximum layer in which each vector appears in the HNSW index is determined randomly, with an exponentially decaying probability as the considered layer is closer to the top. This mechanism results in a uniform distribution of nodes across all layers of the vector space, with sparser representation of vectors at higher layers. Such a distribution ensures even coverage of the *search space* and guarantees the presence of an entry point at approximately equal distances from any query vector, regardless of its location.

However, when dealing with a **clustered** *query workload*, as introduced in chapter 4, certain regions of the *search space* are queried more frequently than others. Given that the query distribution is non-uniform, it stands to reason that a uniform selection of entry points in the upper layers may not be optimal.

We therefore aim to investigate and exploit this intuition to improve entry point selection. Our objective is to minimize the number of distance computations required to answer a query by leveraging workload information during the selection of upper-layer nodes. Additionally,

we analyze how the recall is affected when workload information is considered for entry point selection.

Other approaches focus on static graph measures to guide entry point selection. One work proposed to use as entry points nodes with high in-degree (*hubs*) [44], while other considers the *Local Intrinsic Dimensionality (LID)* of each nodes, promoting nodes with low LID to entry points [45]. As mentioned in chapter 3, these approaches implicitly assign equal importance to all regions of the graph, and therefore do not try to follow the access patterns.

5.2 Incorporating Workload Knowledge

Without loss of generality, we focus our attention on layer 0 of the HNSW index. This layer contains exponentially more nodes than any other layer and, according to the search algorithm described in [37], it is the only layer for which *ef_Search* exceeds 1. Consequently, the majority of distance computations are performed at this layer, rendering it the most promising target for optimization.

Assuming a clustered workload, as described in chapter 4, some nodes are inherently more likely to be included in the set of *top-K* results compared to others. This is because the *top-K* results are defined as the k^{th} closest vectors to any given query, and queries tend to be concentrated in clusters. Therefore, vectors residing within these clusters are more frequently traversed and are likely to be reached sooner during search.

To illustrate, consider Figure 5.1. Intuitively, vectors situated near the highly queried (trending) regions are more likely to appear among the *top-K* results. Thus, if the search procedure is initiated from one of these nodes (as opposed to a distant orange entry point), the number of hops required to converge is reduced, leading to a lower number of distance computations overall.

The central idea is then to **increase the representation of trending regions in the upper layers of the graph**. By doing so, and while maintaining the same total number of nodes in the upper layers, we can achieve greater entry point density in frequently queried areas (thus enabling the search to start closer to the query vector and consequently to the *top-K* results), at the expense of reducing the representation in less frequently accessed regions.

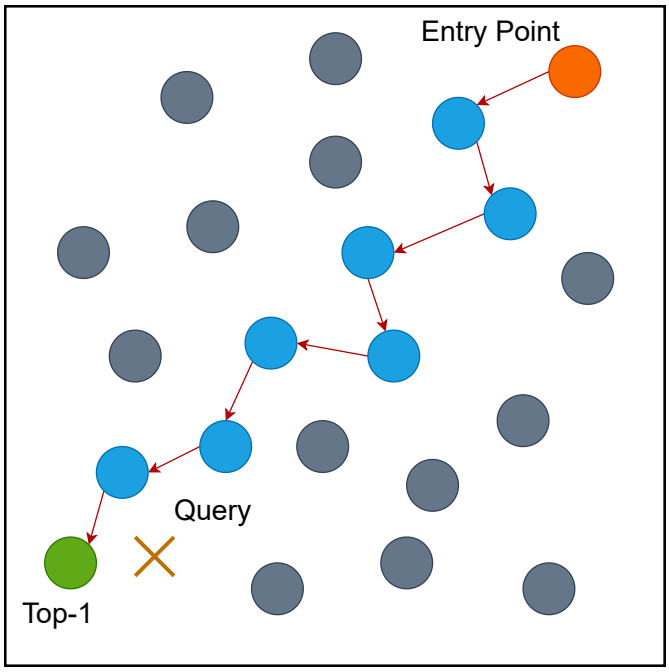


Figure 5.1: Path from the Entry Point to the top-1 result

5.3 Experiments

5.3.1 System Architecture

The system consists of two primary components: the `Python` process - denoted as process A — and the process (or set of processes) within the VDBMS that receives and executes queries from process A, denoted as process B.

For each query, process A initially sends a k-ANN search request to process B. Process B executes the query and returns the results to process A, which records the query recall, the number of distance computations, and the top-1 result.

Subsequently, process A issues a new request using the same query, directing process B to perform a modified search restricted to the HNSW layer 0. The modification depends on the specific experiment.

In subsection 5.3.3, this involves using the previously obtained top-1 result as the entry point. In subsection 5.3.4, it involves setting a fixed entry point.

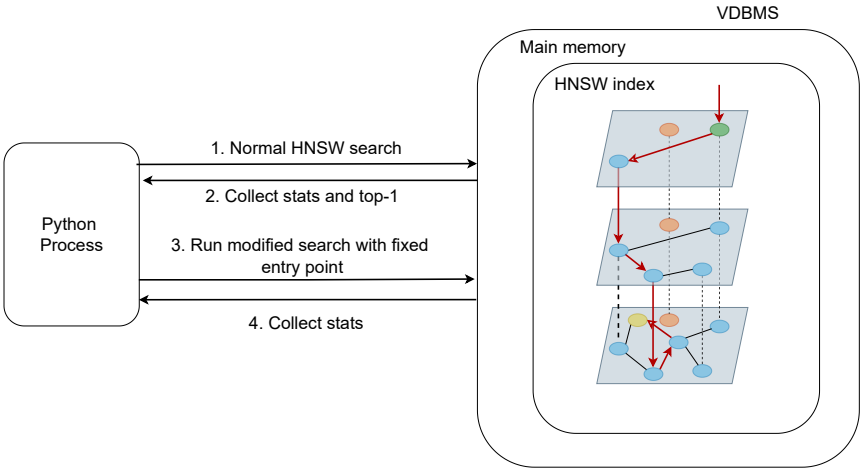


Figure 5.2: System Architecture used to run experiments concerning entry point selection

5.3.2 Experimental Setup

The experiments are conducted for various *ef_Search* and *top-K* settings, using an HNSW index constructed with *ef_Construction* = 300 and $M = 32$. Both the query and base sets are sourced from `http://corpus-texmex.irisa.fr` and `https://nlp.stanford.edu/projects/glove/`, and we utilize the complete set of available queries. Since the top-1 for each query is assumed to be known, there is no need for using a synthesized workload, allowing for a more comprehensive analysis.

5.3.3 Optimal Case Study: Using the Top-1 as Entry Point

To evaluate this hypothesis and to assess whether any algorithm leveraging this intuition could be effective, we conducted a theoretical experiment.

Initially, we execute queries against the HNSW index, recording the number of distance computations in layer 0, the vector returned as the top-1 result, and recall for each query. Subsequently, we re-run these queries, but this time, we forcibly set the entry point in layer 0 to be the top-1 result previously obtained for each respective query. In other words, each query now begins its search from the vector that was identified as its top-1 neighbor in the standard search.

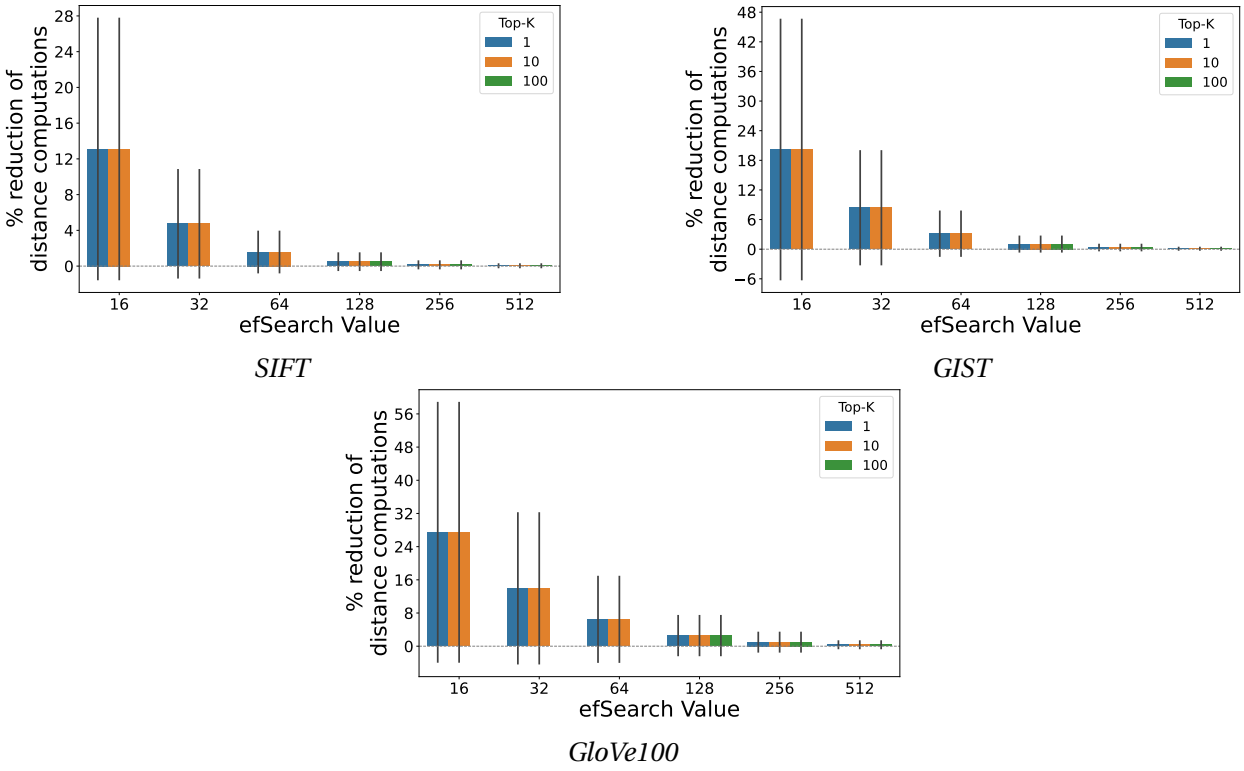
This experiment is purely theoretical, as knowing the top-1 result for a query *a priori* would allow us to return the answer directly in a real top-1 search scenario. Nevertheless, this best-case setup provides insight into the maximal potential improvement achievable by such an approach.

Improvement in number of distance computations

Figure 5.3 presents a comparative analysis, across different *ef_Search* and *top-K* values, of the number of distance computations in layer 0 when the entry point is set to the query's top-1 result ($\mathcal{DC}_{optimal}$) versus when the standard entry point selection is used according to the HNSW algorithm on layer 1 ($\mathcal{DC}$). Specifically, the y-axis represents

$$(\frac{\mathcal{DC}}{\mathcal{DC}_{optimal}} - 1) \times 100$$

averaged over all queries, with standard deviations also presented.



SIFT *GIST*

GloVe100

Figure 5.3: Decrease in distance computations using the top-1 result as entry point compared to default entry point over different *ef_Search* and *top-K* values. Comparison between search in layer 0 while using the top-1 as entry point vs normal search

Firstly, the number of distance computations does not vary with *top-K*. Hence, the improvement rate is consistent across different values. In the HNSW framework, as defined by the search and construction procedures in [37], search complexity is determined primarily by *ef_Construction*, *ef_Search*, and M.

Secondly, for small *ef_Search* values, using the top-1 as the entry point leads to a substantial reduction in distance computations. Depending on the dataset, the improvement ranges from approximately 13% (*SIFT*) to nearly 30% (*GIST*). However, this improvement diminishes

significantly as *ef_Search* increases. For all datasets, when *ef_Search* = 128, the advantage drops to between 0% and 5%. The reason is that higher *ef_Search* values cause more vectors to be visited per search, as shown in subsection 6.3.4. As a consequence, a higher *ef_Search* allows a greater number of candidates in the candidate set C (see section 2.4), which means that vectors further from the query are also considered. While this reduces the risk of search becoming trapped in local optima, it incurs the cost of exploring broader regions of the graph. Most of the computation is then spent not on navigating from the entry point to the top-1 region, but rather on exploring its neighborhood. Although starting closer to the top-1 consistently reduces distance computations, when the exploration makes up most of the total number of distance computations, this improvement becomes negligible.

Additionally, the standard deviation is notably high compared to the mean, particularly for low *ef_Search* values, and decreases as *ef_Search* grows. For small *ef_Search* (e.g., 16 and 32), high variance indicates that, for some queries, using the top-1 as the entry point yields little benefit, while for others the benefit is significant. In Figure 5.3, the mean minus the standard deviation even falls below zero, which requires careful interpretation. This does not necessarily indicate cases where the standard entry point yields better results than the top-1 entry point. When saying the top-1 entry point, we mean the entry point used in the experiment, which was the result of one HNSW search. It is not necessarily the true nearest neighbor of the query, as the result from HNSW search is not guaranteed to be correct. Since HNSW is a proximity graph, starting at the top-1 node ensures that the first *ef_Search* elements inserted into C and W are very likely to be the closest to the query. Consider the scenario where *ef_Search* = 16 and M = 32: it is highly probable that both C and W reach their maximum capacity after adding the immediate neighbors of the entry point. If the graph closely approximates an RNG, the neighbors of the top-1 node are also likely to be near the query. If second-order neighbors are reached, it is improbable that further nodes will enter both sets, and the search thus terminates when all elements in C are exhausted or when the worst element in W is closer to the query than the best in C.

As *ef_Search* increases, so does the capacity of both sets, resulting in more elements being included at the start of search, regardless of their proximity to the query. Consequently, nodes farther from the query enter the candidate set, facilitating an expansion of the search into other graph regions, since it easier for these elements to be replaced in both queues. Thus, even when starting at the top-1 node, the search may traverse additional regions, diminishing the benefit compared to standard search algorithms, since distance computations are distributed over a broader graph area and not concentrated solely on improving the entry point to top-1 path.

The high standard deviation, especially at low *ef_Search*, warrants further explanation. In some cases, the standard search may already select the top-1 node as the entry point or select a node only a few hops away. In other instances — because each HNSW layer is a NSW graph — they contain both long and short range connections, allowing even distant entry points to connect quickly with the top-1 region. If this happens, both methods traverse nearly identical paths, and thus $\frac{\mathcal{DC}_{optimal}}{\mathcal{DC}} \approx 1$. Otherwise, the ratio will slightly exceed one.

Moreover, in some queries, standard search may not retrieve the true top-1, whereas the modified strategy, by starting closer to the true top-1, may escape a local minimum and find the correct neighbor. While this scenario improves recall, it may also require more distance computations than standard search and thus result in a lower $\frac{\mathcal{DC}_{optimal}}{\mathcal{DC}}$ value.

Nevertheless, starting the search at the top-1 node, particularly for low *ef_Search*, typically yields a substantial reduction in distance computations, with improvements that can considerably exceed the mean. These varied scenarios underline the significant per-query variability of $\frac{\mathcal{DC}_{optimal}}{\mathcal{DC}}$, and account for the high standard deviation observed, especially at lower and intermediate *ef_Search*.

Improvement in Recall

We also evaluate the recall to determine whether initiating the search closer to the top-1 result leads to a measurable improvement. For each query, we employ a procedure analogous to the one described for distance computations. Rather than counting distance computations, we record Recall@K for various *top-K* values under standard search ($\mathcal{R}^K$). When queries start search in layer 0 from their respective top-1 result, we similarly collect Recall@K, denoted $\mathcal{R}^K_{optimal}$.

The y-axis on the recall improvement graphs is defined as

$$(\frac{\mathcal{R}^K_{optimal}}{\mathcal{R}^K} - 1) \times 100$$

In cases where $\mathcal{R}^K = 0$, we set the ratio to 1. In such situations, $\mathcal{R}^K_{optimal}$ could be either zero or nonzero: if both are 0, then no improvement in recall has occurred, justifying the ratio of 1; if $\mathcal{R}^K = 0$ and $\mathcal{R}^K_{optimal} > 0$, this would indicate some improvement, but we conservatively treat it as 0 for consistency, still resulting in a value of 1.

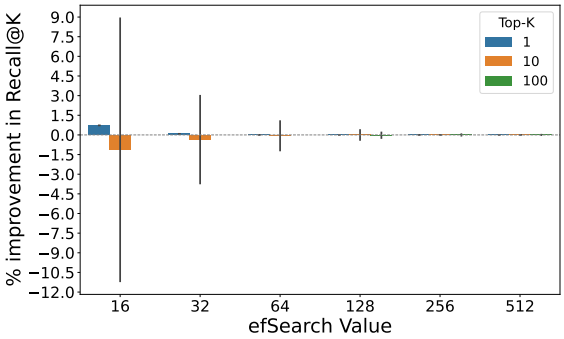
For top-1 queries, both $\mathcal{R}^K$ and $\mathcal{R}^K_{optimal}$ are binary (either 0 or 1), so the above division does not fully capture improvements. Therefore, for top-1, we use the following:

$$(\frac{\sum_{q \in T} \mathcal{R}^K_{optimal}}{\sum_{q \in T} \mathcal{R}^K} - 1) * 100$$

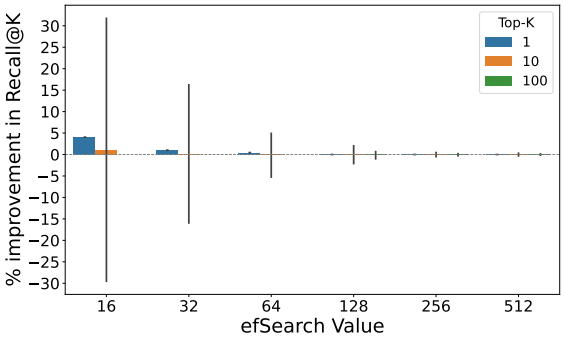
where T denotes the *query dataset* used in the experiment. As a result, for top-1, standard deviation is not reported.

Figure 5.4 illustrates recall improvement, as defined above, across various *top-K* and *ef_Search* values. The main finding is that, for the tested configurations, starting the search at the top-1 location does not lead to a notable change in recall.

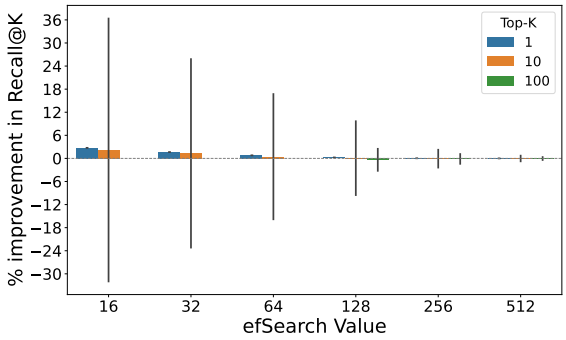
For *top-K* = 1, recall can only be 0 or 1. Since the modified search begins at the top-1 result of each query under the standard search, only two scenarios are possible:



SIFT



GIST



GloVe100

Figure 5.4: Improvement in Recall@K when using the top-1 result as entry point compared to the default entry point over different *ef_Search* and *top-K* values. Comparison between search in layer 0 while using the top-1 as entry point vs normal search

1. The top-1 under standard search is actually the true top-1 neighbor ($\mathcal{R}^K = 1$).
2. The top-1 under standard search does not correspond to the true top-1 neighbor ($\mathcal{R}^K = 0$).

In case (1), by definition, the top-1 result is the dataset vector nearest to the query. Since it is used as the entry point, it is immediately inserted into C and W. Given its minimal distance to the query, it will not be removed and will be returned as the top-1 result. Hence, $\mathcal{R}^K_{optimal} = 1$.

In case (2), since the search begins closer to the true top-1 than in the standard search, there is a possibility - although not guaranteed - that the search will discover the true top-1, yielding $\mathcal{R}^K_{optimal} = 1$. Otherwise, $\mathcal{R}^K_{optimal} = 0$. Consequently, for *top-K* = 1, the y-axis values are restricted to $\mathbb{R}^+_0$.

However, for *top-K* > 1, it is possible that starting from the standard top-1 result does not lead to all the true nearest neighbors being found, and consequently, in some cases, negative values are observed.

Additionally, as *ef_Search* increases, the standard deviation and improvement in Recall approach 0. This is explain by: (a) standard search already yields recall close or equal to 1, making an improvement difficult or impossible to attain and (b) a large portion of the graph is already visited, so the nodes visited by standard search and the modified search have considerable of overlap, hence are likely to return the same results.

5.3.4 Impact of Hierarchy in HNSW querying

Prompted by the preceding somewhat unexpected observations, we also examine the converse approach: evaluating search performance when initiated from a random entry point. This enables us to assess the significance of the upper layers in reducing distance computations and enhancing recall.

To this end, we fix the entry point in layer 0 to a single vector for all queries, as is done in NSW. Results are compared in the same manner as in subsection 5.3.3. The difference here is that, for standard search, distance computations across all layers (rather than only layer 0) are considered. Since our focus is on the impact of the upper layers, we also account for the computations needed to identify the most suitable entry point using the HNSW search algorithm, reason for which we consider all distance computations. Therefore, we are effectively comparing the performance of NSW with HNSW.

Improvement in number of distance computations

As displayed in Figure 5.5, an opposite trend emerges relative to the top-1 experiment. For small *ef_Search* values, there is a non-negligible increase in the number of distance computations,

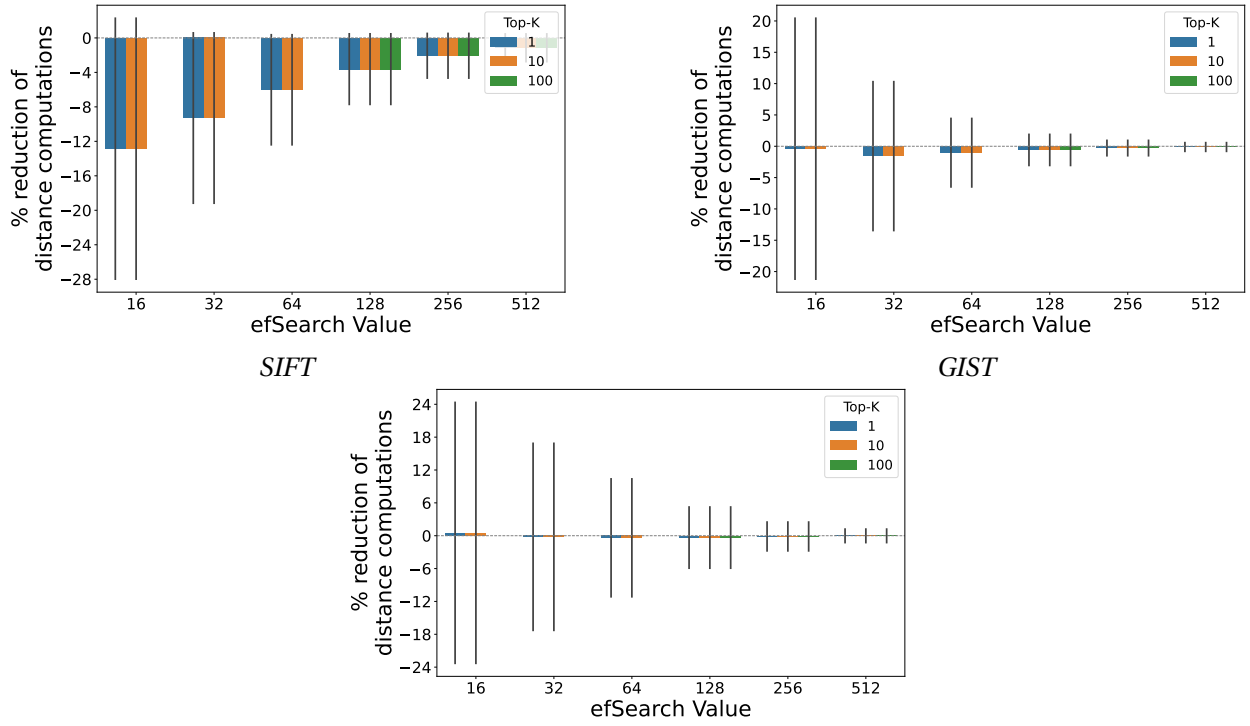


Figure 5.5: Decrease in distance computations using a fixed entry point compared to default entry point over different *ef_Search* and *top-K* values. Comparison between search in layer 0 while using a random, fixed entry point with normal search using all layers

since most of the computation is spent navigating from the entry point to the result rather than exploring the result's neighborhood. Under these conditions, starting closer to the result (as in HNSW) greatly reduces the search cost, and the additional distance computations performed in the upper layers are justified.

An exception is observed for the *GloVe100* dataset at *ef_Search* $= 16$, which likely results from the increased difficulty of this dataset. For this specific configuration, the reduction in layer 0 computations when using the entry point selected during search does not compensate for the extra computations needed to locate that entry point. Also, for *GIST*, using NSW causes a larger increase in distance computations for *ef_Search* $= 32$ relative to *ef_Search* $= 16$, and likewise for *GloVe100* at *ef_Search* $= 64$ versus *ef_Search* $= 32$. Given the high variance and marginal numerical difference, we do not consider these effects to be statistically significant.

However, as *ef_Search* increases, the majority of distance computations are performed in layer 0 during the exploration of result neighborhoods, reducing the positive impact of upper layers. In these cases, starting at different entry points offers a minimal reduction in overall computations, compared to starting directly in layer 0.

The search dynamics and reasons for a high standard deviation follow the same logic outlined in subsection 5.3.3, with the only difference that HNSW is more likely to start closer to an optimal entry, while NSW is more likely to start further away.

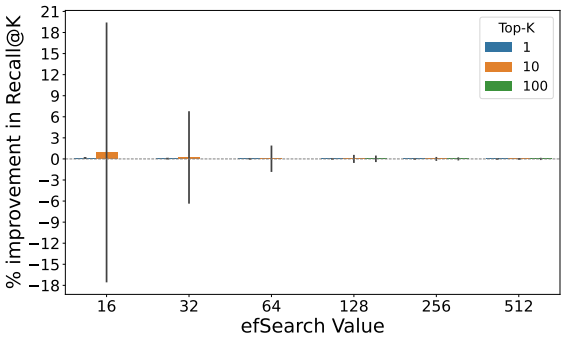
These results are also backed by other works such as [44].

Improvement in Recall

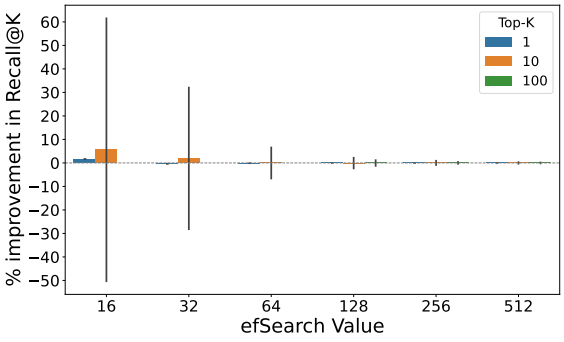
Figure 5.6 presents the corresponding recall results. Once again, the impact on recall appears negligible, particularly for higher *ef_Search* values. The explanation remains consistent with previous observations: as the level of graph exploration increases, the results retrieved by NSW approximate those obtained with HNSW. Conversely, when exploration is limited, the probability of becoming trapped in local minima increases, which leads to substantial variability in results across queries and, consequently, a larger standard deviation.

5.4 Conclusion

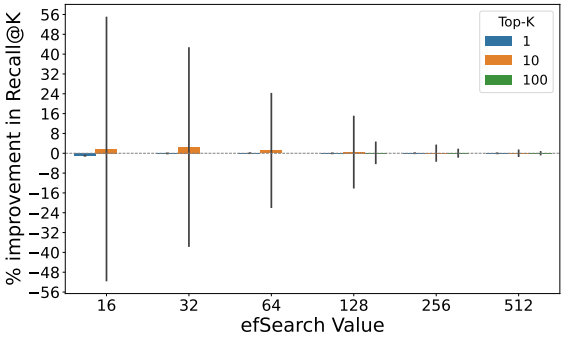
In this chapter, we investigated an alternative approach to promoting nodes to upper layers — distinct from the default random selection — by exploiting workload knowledge. The underlying idea is to disproportionately represent graph regions that experience high query accesses, while underrepresenting those less frequently accessed. This aims to start the search process, on average, closer to the query vector, thereby reducing the number of hops required to retrieve the



SIFT



GIST



GloVe100

Figure 5.6: Improvement in Recall@K when using a fixed entry point compared to the default entry point over different *ef_Search* and *top-K* values. Comparison between search in layer 0 while using a random, fixed entry point with normal search using all layers

result.

To evaluate the effectiveness of this intuition, we conducted an optimal case study: we assessed the impact on recall and distance computations when the top-1 result of a query — retrieved using the standard HNSW search — was employed as the entry point at layer 0, rather than the default entry point. This framework allowed us to estimate the maximum possible improvements from strategies that overrepresent frequently accessed regions of the graph.

Our findings demonstrate that the proposed approach is effective primarily for low *ef_Search* values. In such cases, where graph exploration is limited, the reduction in hops translates directly into a significant decrease in distance computations. However, as *ef_Search* increases, this advantage quickly diminishes, since broader exploration minimizes the relative contribution of the improved entry point when compared to the total number of distance computations. A similar pattern was observed for recall: as *ef_Search* grows, paths from the standard entry point and from the top-1 are increasingly similar, retrieving identical or nearly identical results. At lower *ef_Search* values, however, searches are more prone to becoming trapped in local minima, which explains the high standard deviation observed in recall.

We then extended this analysis to evaluate the role of hierarchy in HNSW. Specifically, we compared the number of distance computations between standard HNSW searches and searches initialized at a fixed entry point in layer 0. Here, we found similar results: for low *ef_Search* values, the presence of upper layers substantially reduced the number of distance computations (depending on the dataset), enabling HNSW to significantly outperform NSW. The effects on recall mirrored this trend.

In response to the research questions posed in section 3.1:

LATENCY_Q1 The maximum observed improvement in reducing distance computations and recall occurs at low *ef_Search* values, reaching nearly 30% for the *GloVe100* dataset, with an associated recall improvement of approximately 5%. However, as *ef_Search* increases — which is necessary for achieving high recall targets — these improvements rapidly diminish until they become negligible.

LATENCY_Q2 Given the negligible improvements for high recall queries (which necessarily rely on large *ef_Search* values), we did not pursue further strategies for node promotion to upper layers, as our primary focus lies on high-recall search performance, where maximal improvements remain insignificant.

LATENCY_Q3 Given the negligible improvement for high recall queries, we did not develop any method, and therefore there was no method to empirically evaluate.

Chapter 6

Memory: Reducing the memory footprint

In this chapter, we investigate our second sub-problem - the main contribution of our work - which concerns the HNSW memory requirements. We begin by refining the problem statement, building on the broader introduction provided in chapter 3. Subsequently, we provide a detailed analys of both our baseline and our proposed methods, showing each method's advantages and disadvantages and precisely defining their inner workings. Additionally, we present and interpret a comparative experimental analysis, assessing our methods in multiple metrics both in terms of efficacy and adaptation to multiple search and workload configurations, and comparing them to another hybrid indexing technique, DiskANN [24].

6.1 Problem Statement

As discussed in chapter 3, the memory requirements of an HNSW index grow rapidly with dataset size, eventually making it infeasible to store the entire index in-memory. HNSW demands all vectors residing in RAM to ensure efficient distance computations. This is essential, since each search requires multiple vector comparisons for traversing the graph across different layers to converge on a result. Consequently, there is a hard upper limit on the dataset sizes that can be indexed by HNSW. If the vectors exceed the available main memory, the standard HNSW approach [37] can no longer be applied. Therefore, addressing this shortcoming is of vital importance to the scalability of HNSW indexes.

Nevertheless, when the query workload is **clustered** — as introduced in chapter 4 — certain regions of the *search space* are queried more frequently than others. This non-uniform query distribution implies that some vectors may rarely, if ever, be accessed and therefore do not need to reside in memory. Identifying such infrequently accessed vectors and offloading them to

disk would ease memory constraints, potentially without adversely affecting recall or latency. Generalizing this intuition, due to workload bias, some nodes are far more likely to be visited during search, and this pattern can be explored.

To address the memory bottleneck of an in-memory HNSW index, we propose the use of an adaptive caching mechanism. The cache, constrained by a fixed capacity denoted as *memory budget*, cannot store all vectors simultaneously. Given known storage requirements per vector, the value of *memory budget* allows us to compute the maximum number of vectors that fit in the cache for a given dataset. All remaining vectors are relegated to disk storage. When a disk page can accommodate multiple vectors, a policy for grouping vectors onto the same pages becomes necessary. If a single vector spans several pages, it is natural to store these pages contiguously, though other strategies may be used. For this thesis, unless otherwise specified, we make no assumptions regarding the layout of vectors on disk, ensuring that our proposals remain compatible with any storage scheme. Nevertheless, storage strategies can significantly impact the latency of fetching vectors (or batches) from disk. We leave the exploration of the optimal storage strategy for these methods as future work.

Crucially, we assume that the graph metadata (such as neighbor lists) and all vectors belonging to the upper layers always reside in memory. These components occupy a relatively small portion of the total storage, and thus have negligible impact on the cache size. In other words, when we refer to the cache size as *memory budget*, we actually mean *memory budget* plus up to the size taken to host all vectors in layer 1. Since vectors in layers above 1 form a subset of those in layer 1, this definition accounts for all layers. In practice, some vectors in the upper layers may already be present in the cache because they were chosen to be cached by the used method.

In this chapter, we introduce and analyze several *cache priority policies*. As described in section 2.5, these policies assign priorities to each node. Our proposals consider the **query workload** in determining these priorities, but workload-agnostic policies (e.g., using node in-degree) are also conceivable.

Additional Notation To formally define our objective, we introduce further notation building on section 3.3.

Let:

- θ denote the user-defined buffer cache *memory budget* and Θ denote the maximum number of vectors that can be stored in the cache for *memory budget* = θ.
- $B \subseteq \mathcal{V}$, $|B| = \Theta$ denote the subset of cached nodes residing in the cache. Vectors in $\mathcal{V} - B$ are stored on disk and loaded as required. The set B consists of the Θ highest-priority nodes per the selected policy.
- $SG_B = (B, \mathcal{E}_B)$ denote the subgraph induced by the cached nodes, with $\mathcal{E}_B = \Big\{(s,t) \in$

$\mathcal{E} \mid s,t \in B\Big\}$.

- $UVis_{\mathcal{Q}}^{TRAIN}, UVis_{\mathcal{Q}}^{TEST}$ denote the sets of unique nodes visited during search for all train (resp. test) queries in a sample $q \sim \mathcal{Q}$. That is, for a set of queries T (training or testing), $\bigcup q \in Tvisited(q)$.
- $\#visited_{\mathcal{Q}}^{TRAIN}(v), \#visited_{\mathcal{Q}}^{TEST}(v)$ denote the frequency with which a node $v \in \mathcal{V}$ is visited by train (resp. test) queries from $\mathcal{Q}$. For a query set T, $\#visited_{\mathcal{Q}}^{TRAIN}(v) = \sum_{q \in T} |v \cap visited(q)|$.
- $\#Visited_{\mathcal{Q}}^{TRAIN}, \#Visited_{\mathcal{Q}}^{TEST}$ denote arrays of size $|\mathcal{V}|$ such that $\#Visited_{\mathcal{Q}}^{TRAIN}[i] = \#visited_{\mathcal{Q}}^{TRAIN}(v_i)$ for all $v_i \in \mathcal{V}$ (and similarly for test). Each vector has a contiguous id from 0 to $|\mathcal{V}| - 1$.

Goal The goal is to devise an algorithm that smartly selects a subset of dataset vectors to reside in memory, with the remainder stored on disk. We aim to find the set B that maximizes intersection with the set of nodes visited during search:

$$argmax_B \, |B \cap visited(q)| \tag{6.1}$$

for a query $q \sim \mathcal{Q}$.

We focus primarily on HNSW, as it is among the most widely used graph-based indexes. However, the generality of our formulation allows for the proposed methods to be extended to any graph-based index to achieve intelligent caching. Evaluation of these methods on other index structures is left as future work.

Unlike traditional buffer caches, in which the newly read item from disk takes the place of some item in the cache, we assume a static cache: the contents are determined from training data (e.g., queries observed over a specific interval) and remain fixed until the policy is explicitly re-run. Consequently, this enables us to focus on a simplified, temporally static scenario. Detecting workload shifts and triggering cache re-evaluation is also left as future work. Even though this assumption is convenient, there is logic behind it. Since queries may visit many nodes, a dynamic policy could cause a single sequence of outlier queries to evict frequently used vectors from cache, resulting in excessive disk I/O for the majority of the queries. Constantly adapting the cache's elements could prove counter-productive, as it would overfit to the nodes visited by a single query. Furthermore, some nodes may be extremely unlikely to be accessed (e.g., nodes with an in-degree of one, only reachable from a single parent among, potentially, billions). Such nodes, even if occasionally accessed, should not displace more frequently visited ones.

For these reasons, we assume that when a node required during search is not in cache, it is read from disk as needed, used for the search, and then its memory is released.

6.2 Proposed Methods

In this section, we present and explain the various methods explored in our experiments. We refer to them as "methods" to emphasize their algorithmic nature, but they can also be interpreted as cache priority policies. We define our baseline method and justify its selection. Furthermore, we describe additional approaches considered during the research phase but ultimately excluded from this thesis, providing a rationale for their omission.

6.2.1 Baseline: MFU

Given our assumption of a **clustered** *query workload*, certain vectors are more likely to be visited during the search for a query $q \sim \mathcal{Q}$. Therefore, for a set of training queries (and $visited(q)$ computed for each), our baseline approach is to select the Θ vectors appearing most frequently in $visited(q)$ across all training queries. The caching priority of each vector is thus the number of times it was visited in searches induced by the training set. This aligns with the Most Frequently Used (MFU) policy described in section 2.5. An illustration of this policy is presented in Figure 6.1, where the number inside each node indicates the total visit count from all training queries ($\#visited_{\mathcal{Q}}^{TRAIN}(v)$), while the node color distinguishes whether or not it is cached.

While intuitive, this baseline method exhibits several limitations:

Caching Nodes with the Smallest Frequency Consider the minimum visit frequency for any cached node, 12 (Figure 6.1). If $\Theta = 9$, not every node with frequency 12 can be retained in memory, and the policy must choose arbitrarily among them. However, among nodes with equal frequency, some may in fact be more likely to be visited. For example, nodes with higher in-degree, as observed in [44], might be accessed more often in practice. Therefore, further prioritization within ties could yield better results.

Ignorance of Graph Structure The policy does not account for the graph structure. For instance, a cached node with frequency 12 may have no neighbors in memory. Consequently, any search that reaches this node would still require loading at least one neighbor from disk. Ideally, the cache should enable most queries to be satisfied from memory alone. If achieved, an index occupying only a fraction of the original size could potentially answer queries with the same recall and latency as a fully in-memory index. However, because the MFU policy is agnostic to graph connectivity, the induced subgraph SG_B may be highly fragmented, thus limiting the likelihood that B contains all (or most) of the nodes traversed during any search. This issue is also related to the aforementioned problem.

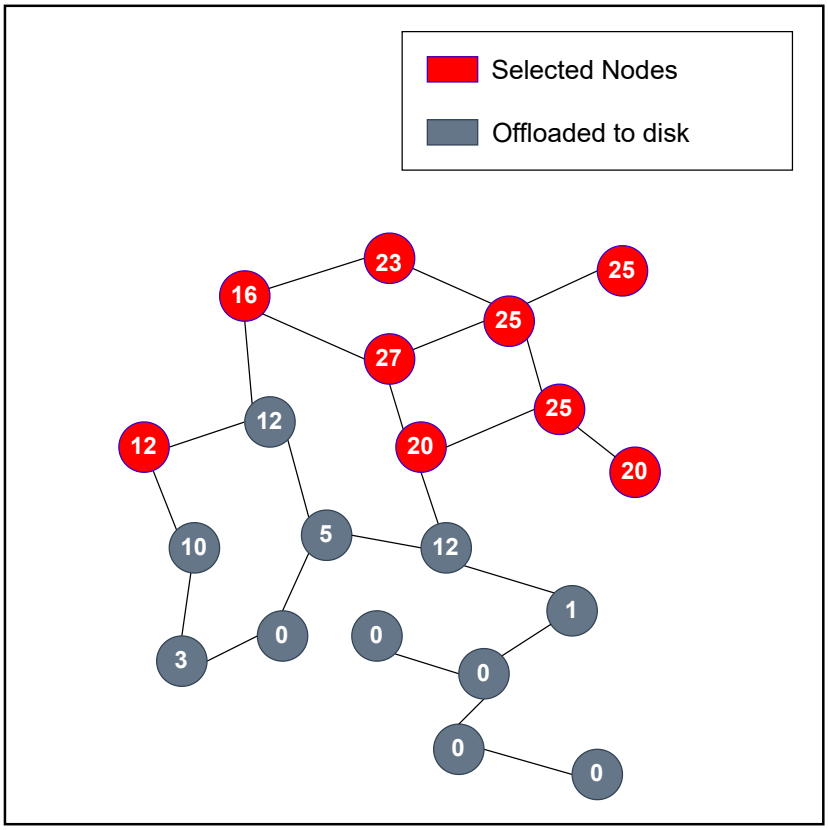


Figure 6.1: Baseline method for $\Theta = 9$

Lack of Generalization Suppose $\Theta > |UVis_{\mathcal{Q}}^{TRAIN}|$. In this scenario, all nodes from $UVis_{\mathcal{Q}}^{TRAIN}$ would be cached, and the remaining $\Theta - |UVis_{\mathcal{Q}}^{TRAIN}|$ cache slots would be assigned to unused nodes. Since these unvisited nodes all have a frequency of zero, they are chosen at random from $\mathcal{V} - UVis_{\mathcal{Q}}^{TRAIN}$. For large datasets, $UVis_{\mathcal{Q}}^{TRAIN}$ often represents only a small fraction of the total nodes. Moreover, in practical use, incoming queries are unlikely to exactly reproduce those in the training set: new queries may differ slightly even when referring to the same target (e.g., slight variations in e-commerce search text queries or embeddings produced by LLMs for different textual inputs describing the same concept). Since embeddings are sensitive to input text, seemingly minor differences may result in different paths being traversed within the HNSW graph. Given that our ultimate goal is to be able to serve a majority of queries from the same workload with minimal to no performance degradation, there will be a significant portion of nodes that will be visited by incoming queries and were not visited by the train query set. Thus, policies must generalize beyond the training set to ensure that previously unseen — but closely related — queries can also be efficiently served from the cache. This only becomes feasible when $\Theta > |UVis_{\mathcal{Q}}^{TRAIN}|$.

We selected this method as our baseline since it is intuitive and represents a natural starting point for the problem. Additionally, MFU is a well-known caching algorithm, extensively analyzed across various fields [15, 40, 52]. The method also presents several benefits:

Memory and Runtime Efficiency Given $\#Visited_{\mathcal{Q}}^{TRAIN}$, this method requires only selection of the Θ vectors with the highest $\#visited_{\mathcal{Q}}^{TRAIN}(v)$ values. Its memory complexity is $\mathcal{O}(\Theta)$, requiring only an array of this size. Practically, the algorithm maintains a min-heap initialized using the first entry of $\#Visited_{\mathcal{Q}}^{TRAIN}$, paired with its corresponding vector ID. For each entry $\#Visited_{\mathcal{Q}}^{TRAIN}[k]$ with $0 < k < |\mathcal{V}|$, the algorithm adds the element to the min-heap if it has less than Θ elements or if it is larger than the heap root, in which case the root gets replaced; otherwise, it is discarded. The final set B is thus

$$\left\{ v_j \mid \forall (j, \#visited_{\mathcal{Q}}^{TRAIN}(v_j)) \in min - heap \right\}$$

The time complexity is $\mathcal{O}(|\mathcal{V}| \log \Theta)$: traversing $\#Visited_{\mathcal{Q}}^{TRAIN}$ of size $|\mathcal{V}|$, with each heap operation (pop + push) taking $\mathcal{O}(\log \Theta)$ in the worst case. Empirically, the method is extremely fast, as it involves sequential accesses to two contiguous arrays.

Lastly, the method is simple to implement and easy to reason about.

6.2.2 Expand from the Visited Set (EVS)

Building upon the observations outlined in subsection 6.2.1, we propose the Expand from the Visited Set (EVS) method.

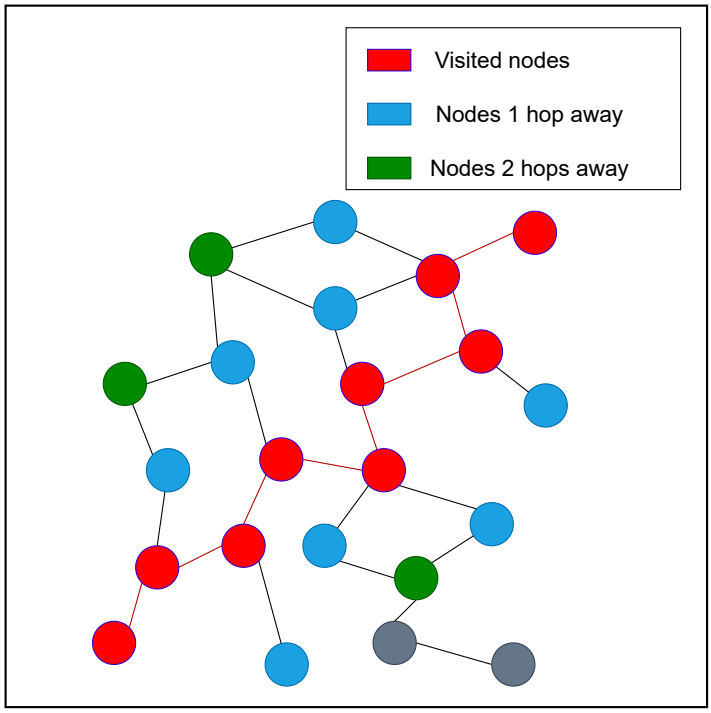


Figure 6.2: Expanding from the visited set. The caching priority of unvisited nodes is equal to the of the number of hops they are from any visited node

Firstly, if $\Theta \leq |UVis_Q^{TRAIN}|$, this method selects the same set B as the Baseline approach. In this scenario, all benefits, limitations, and complexity considerations are identical to those discussed in subsection 6.2.1. Thus, the subsequent discussion focuses on the case where $|UVis_Q^{TRAIN}| < \Theta$, unless stated otherwise.

This method initiates a Breadth-First Search (BFS) starting from the set $UVis_Q^{TRAIN}$. Since a standard BFS begins from a single root, we conceptually introduce an imaginary root node with in-degree zero, connected by one outgoing edge to every node in $UVis_Q^{TRAIN}$.

In Figure 6.2, we illustrate this method. Initially, $B = UVis_Q^{TRAIN}$. At each BFS expansion step, the current node is added to B. If $|B| = \Theta$, the procedure terminates and returns B. Otherwise, the neighbors of the current node are added to the BFS queue.

Caching priorities are defined as follows: if $\#visited_Q^{TRAIN}(v_i) > \#visited_Q^{TRAIN}(v_j)$, node v_i takes precedence over v_j. If $\#visited_Q^{TRAIN}(v_i) = \#visited_Q^{TRAIN}(v_j) \neq 0$, both nodes have equal priority, and ties are resolved randomly. This can only happen if $|UVis_Q^{TRAIN}| > \Theta$, and therefore this decision is the same made by the baseline. If $\#visited_Q^{TRAIN}(v_i) = \#visited_Q^{TRAIN}(v_j) = 0$, priority is determined by the inverse of the shortest hop distance from each node to the closest node in $UVis_Q^{TRAIN}$. If nodes have both zero frequency and equal minimum distance to $UVis_Q^{TRAIN}$, selection among them is random.

This method presents three main improvements over the baseline:

1. **Informed Selection:** When $|UVis_{\mathcal{Q}}^{TRAIN}| < \Theta$, the method does not select the remaining nodes in B at random. Instead, it expands outward from the observed visited nodes. Given a clustered query workload, nearby vectors are likely to be accessed by similar or new queries, justifying this expansion. As all queries (train and the upcoming queries) are sampled from $\mathcal{Q}$, unseen but similar queries will probably traverse nodes in close proximity to those already encountered.
2. **Navigable Memory Subgraph:** By expanding from known visited nodes, this approach tends to form a contiguous, navigable SG_B. This increases the likelihood that the in-memory cache supports direct traversal to answer queries, in which case neither latency nor recall are impacted compared to the completely in-memory index.
3. **Seamless Adaptation to Multiple Clusters:** The expansion procedure naturally accommodates and preserves the clustered structure present in the workload, extending cache coverage efficiently over multiple dense regions if needed.

The method employs two data structures: a set for all visited elements, and a First In First Out (FIFO) queue for BFS traversal. The BFS stops once Θ unique nodes have been popped from the FIFO, including all nodes in $UVis_{\mathcal{Q}}^{TRAIN}$ to initialize a FIFO queue. For each visited node, all unvisited neighbors (up to $2 \cdot M$ in layer 0) are processed and added to the queue. Both structures perform the same inserts, but the queue additionally pops one element per BFS iteration. The visited set is initialized as $UVis_{\mathcal{Q}}^{TRAIN}$, with new neighbors added at each step, so the memory complexity is $\mathcal{O}(\min(|\mathcal{V}|, 2 \cdot M \cdot \Theta))$, which simplifies to $\mathcal{O}(\min(|\mathcal{V}|, M \cdot \Theta))$. The queue has identical memory requirements. The test-and-set operation for visited nodes and queue insertion both take $\mathcal{O}(1)$ time. Additionally, this method still needs to run the baseline method first. Thus, the time complexity of the algorithm is $\mathcal{O}(\min(2 \cdot M \cdot \Theta, |\mathcal{V}|) + |\mathcal{V}| \cdot \log \Theta) = \mathcal{O}(\min(M \cdot \Theta, |\mathcal{V}|) + |\mathcal{V}| \cdot \log \Theta)$. In practice, the actual running time is often better, since nearby nodes may share many neighbors, and most times only a portion of the graph is visited, resulting in fewer total queue insertions.

As shown, this approach incurs higher computational and memory costs than the baseline due to the BFS traversal. Nonetheless, as we demonstrate in section 6.3, it consistently outperforms the baseline in key metrics, when $\Theta > |UVis_{\mathcal{Q}}^{TRAIN}|$. The main problem of this method - in addition to those presented in the baseline method when $\Theta \leq |UVis_{\mathcal{Q}}^{TRAIN}|$ and lack of generalization ability compared to other presented methods - is:

No distinction among equidistant nodes A significant limitation of this method is the inability to distinguish between nodes at the same distance from the closest node in $UVis_{\mathcal{Q}}^{TRAIN}$ with zero visit frequency. The number of nodes at distance k expands exponentially as k increases. Suppose achieving Θ total nodes requires including all nodes k hops away from the starting set, with the remainder chosen from among those $k+1$ hops distant. The selection among nodes

at distance $k+1$ is completely random, despite the likelihood that some are more frequently accessed. Thus, no mechanism exists to prioritise more promising candidates among these equidistant nodes, potentially reducing cache effectiveness.

6.2.3 Expand from the Visited Set In-Degree (EVSI)

Expand from the Visited Set In-Degree (EVSI) is a variant of the EVS method, differing in its approach to prioritizing nodes for caching when $\Theta > |UVis_{\mathcal{Q}}^{TRAIN}|$. Specifically, when selecting nodes located k hops from the nearest node in $UVis_{\mathcal{Q}}^{TRAIN}$, EVSI prioritizes nodes based on their in-degree. This adaptation leverages the observation that nodes with higher in-degree are statistically more likely to be visited during search [44].

All other aspects of the analysis provided in subsection 6.2.2 applies to this method. The key distinction from EVS lies in the cache prioritization as follows: If $\#visited_{\mathcal{Q}}^{TRAIN}(v_i) = \#visited_{\mathcal{Q}}^{TRAIN}(v_j) = 0$ and both v_i and v_j have the same shortest path distance to the closest node in $UVis_{\mathcal{Q}}^{TRAIN}$, then, under EVSI, cache priority is determined by in-degree rather than by random selection.

6.2.4 Personalized PageRank (PPR)

Personalized PageRank (PPR) is a well-established algorithm with widespread applications, including web search, social network analysis, recommendation systems, graph neural networks, community detection, and bioinformatics [26, 29, 56, 61, 68, 75]. It is a variant of the renowned *PageRank* algorithm [49], originally developed by Google to rank web pages. PPR was introduced in the same foundational work as *PageRank*. Over time, the algorithm has found applications in many diverse fields far different from its original scope [20, 26].

Both *PageRank* and PPR can be described using the *random walker model*. In this model, a random walker starts at a node in a directed graph and, at each step, hops uniformly at random to one of its neighbors. This process forms a *Markov Process* [12], where the next state (the neighbor chosen) depends only on the current state (the present node).

The *PageRank* score of a node corresponds to the stationary probability that the random walker occupies that node at an arbitrary step, after infinitely many steps. To ensure that a unique stationary distribution exists, *PageRank* requires the Markov process to be aperiodic and irreducible. However, standard PageRank encounters issues in certain graph structures. For example, if a node has out-degree zero — called *rank sink* [49] — the random walker becomes trapped, eventually concentrating all score mass at that node. Another problem arises with cycles: consider two nodes a and b, where a has two in-edges (one from b and another from r, the rest of the graph), a single out-edge to b, and b only connects back to a. In this situation, once the random walker enters the $a \leftrightarrow b$ cycle, it never exits, and all probability mass is eventually

concentrated evenly between a and b. Both of these situations are represented in Figure 6.3.

Figure 6.3: 2 examples of rank sinks in *PageRank*

To address such problems, *PageRank* incorporates a *teleportation* mechanism: with probability $1-\alpha$, the walker jumps to any node in the graph (chosen uniformly), and with probability α, it transitions to a neighbor, as explained before. This approach breaks cycles and prevents rank sinks from absorbing all the score mass. The parameter α is tunable, with 0.85 being a common choice in most applications [3, 31, 49].

PPR modifies this by introducing a personalization vector. Instead of teleporting uniformly at random, and instead of starting from each node with equal probability, both the start and teleportation operations are informed by a user-defined probability distribution over the nodes: a *personalization vector* of length $|\mathcal{V}|$, ψ. Otherwise, the algorithm is identical to standard PageRank.

The formal definition of the PPR score for node $v_i \in \mathcal{V}$ is given by:

$$PPR(v_i) = (1-\alpha) \cdot \psi\,[i] + \alpha \cdot \sum_{v_j \in in(v_i)} \frac{PPR(v_j)}{|out(v_j)|} \tag{6.2}$$

where $in(v_j)$ and $out(v_j)$ denote the sets of in-neighbors and out-neighbors of v_j, respectively, and ψ represents the personalization vector. Henceforth, this notation is used for the personalization vector that random walker models accept as input. PPR can also be written as:

$$\begin{aligned} (I-\alpha P^T)\pi &= (1-\alpha)\psi \\ (I-\alpha(I-L)^T)\pi &= (1-\alpha)\psi \\ (1-\alpha)I\pi + \alpha L^T\pi &= (1-\alpha)\psi \\ \pi &= \alpha\sum_{k=0}^{\infty}(1-\alpha)^k\psi(I-L)^k \end{aligned} \tag{6.3}$$

where $P = D^{\dagger}A,\ L = I-P$, π is the vector with the final PPR scores for all nodes, D is a diagonal matrix containing the out-degree of the graph nodes, $D^{\dagger}$ the Moore-Penrose inverse of D, and A is the adjacency matrix of HNSW's layer 0 graph [8, 17]. While this notation is not explored in the subsection, it will be in subsection 6.2.5.

In our approach, we propose to use PPR with the $\psi = \#Visited_{\mathcal{Q}}^{TRAIN}$, normalized to form a

probability distribution. Each node v_i is then assigned a cache priority equal to $PPR(v_i)$. This means that random walks are initiated at nodes proportionally to their visited frequency in the training query dataset. Intuitively, this simulates searches starting from nodes that were actually encountered by past queries. Although random walks are not guided by vector similarity — as in HNSW search — our expectation is that PPR will capture patterns in graph traversal that are relevant for both random walks and real query searches.

A single random walk under the PPR method is depicted in Figure 6.4. Nodes containing numbers correspond to elements of $UVis_Q^{TRAIN}$. A random walk begins at any of those nodes with a probability proportional to its frequency. In Figure 6.4, by chance, it was the green node with .20 inscribed. At each step, with probability α the walk moves to a neighbor - each one with probability 0.(3) - and with probability $1-\alpha$ it teleports to a node chosen according to the personalization vector. The process then repeats recursively.

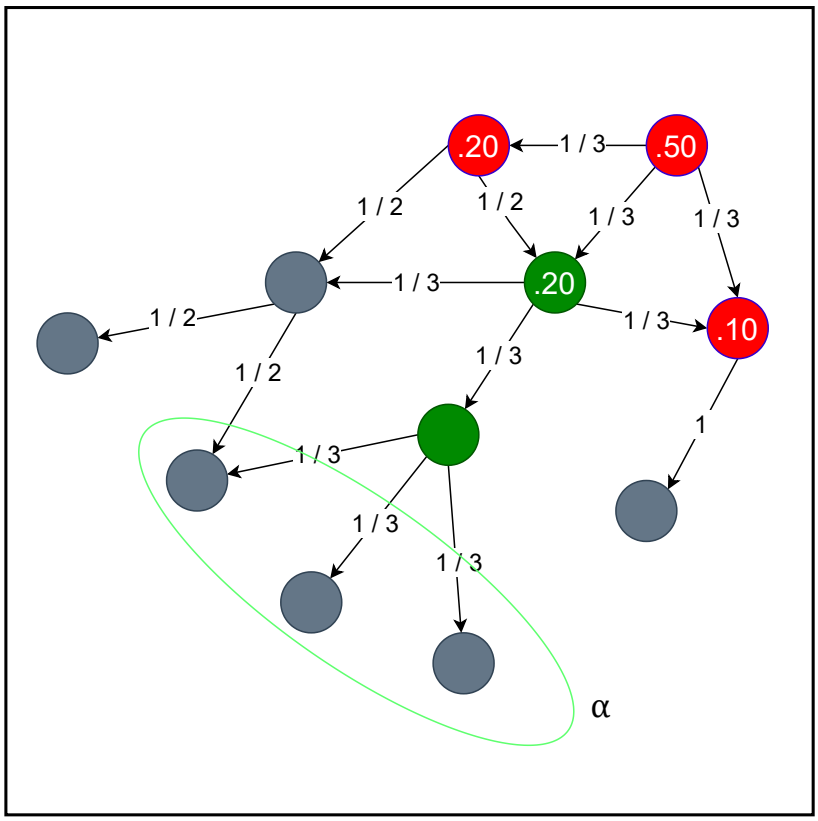


Figure 6.4: PPR random walker model. The numbers inside the nodes represent its personalization value (0 if not present), and values on the edges represent the probability of jumping into the target node from the source node.

This approach offers several advantages over those previously described:

Utilization of Graph Structure and Degree Distribution Because PPR scores are logically based on simulated searches starting from any node in $UVis_{\mathcal{Q}}^{TRAIN}$, they reflect the graph structure and degree distribution. Nodes that are distant (in hops) from any node in $UVis_{\mathcal{Q}}^{TRAIN}$ are unlikely to be visited in random walks initialized from $UVis_{\mathcal{Q}}^{TRAIN}$, and thus receive low PPR scores. Nodes near the observed workload receive higher scores. Moreover, walks starting at a visited node will frequently encounter other visited nodes, so PPR naturally diffuses, generalizes, and smooths visitation frequencies according to both the the initial values ψ and the graph topology.

Connectedness and Generalization The cached nodes tend to form a connected subgraph SG_B, facilitating answering queries primarily or entirely from memory. This is reflected on the results presented in subsection 6.3.5.

The parameter α governs how widely scores diffuse from $UVis_{\mathcal{Q}}^{TRAIN}$: Higher values of α favor long paths, enabling even distant nodes to acquire nonzero scores. Lower values, on the other hand, ensure that scores remain concentrated around $UVis_{\mathcal{Q}}^{TRAIN}$.

Despite its strengths, PPR carries some disadvantages:

Parameter Tuning The teleportation parameter α must be tuned for optimal performance, adding one research variable. We explain how we decided which α to use in subsection 6.3.11.

Computational Complexity Computing PPR is typically more resource-intensive than any of the previously described methods. Nevertheless, efficient algorithms exist with time complexity $\mathcal{O}(\sqrt{\frac{|V|}{\delta}})$, where δ is the mininum PPR threshold we want to detect [33]. Because the HNSW's layer 0 adjacency matrix is sparse, the transition matrix can be stored in $O(|\mathcal{V}| + |\mathcal{E}|)$ space, and it suffices to keep two copies of the PPR vector, incurring additional $O(|\mathcal{V}|)$ memory. Nevertheless, there are methods for approximating these scores that are tailored for large graphs and scale better as graph size increases [33, 34].

In summary, while PPR offers more nuanced cache prioritization by leveraging both workload observations and graph structure, it comes at the cost of greater parameter tuning complexity and increased computational overhead compared to all previous methods. Because PPR is a geometric sum, the decay in influence of one node in other nodes decreases geometrically with the number of hops. As shown in section 6.3, this might not be desirable for the task at hand.

6.2.5 Heat Kernel PageRank (HKPR)

Heat Kernel PageRank (HKPR) [8] is a variant of PPR that satisfies the heat equation, which models the diffusion of heat from regions of high temperature to regions of low temperature. A

formulation of the heat equation that is well-suited to graphs is presented in Equation 6.4:

$$\begin{aligned} & \frac{\partial H}{\partial t} = -LH, H(0) = \psi \\ \Leftrightarrow \quad & H(t) = e^{-tL}\psi \\ \Leftrightarrow \quad & H(t) = \sum_{k=0}^{\infty} \frac{(-t)^k}{k!} L^k \psi \end{aligned} \tag{6.4}$$

Here, $L = I - (D^{\dagger})^{1/2} A (D^{\dagger})^{1/2}$, where D is the diagonal matrix of node out-degrees, $D^{\dagger}$ denotes the Moore-Penrose inverse of D, and A is the adjacency matrix of layer 0 in the HNSW graph. The matrix L, commonly referred to as the graph Laplacian, has several variants with distinct properties. The variant employed here is typically called the symmetrically normalized Laplacian. Its normalization mitigates the bias introduced by nodes with high out-degree by adjusting the influence of each edge according to both the source and target node degrees. Traditionally, L is defined as $D - A$ in the case of undirected graphs, with D representing either the in or out degree diagonal matrix. Another common variant is the random-walk normalized Laplacian, $I - D^{\dagger}A$, in which D is again the out-degree matrix. Since PPR can also be expressed in matrix form using the random-walk normalized L (Equation 6.3), identical normalizations can be applied. This is equivalent to dividing each row of A by the out-degree of the corresponding node, as in Equation 6.2.

Compared to $I - D^{\dagger}A$, if the symmetric Laplacian were used in Equation 6.2, the computation $PPR(v_j)$ would be normalized by $\sqrt{|out(v_j)| \cdot |out(v_i)|}$. Empirically — using the evaluation metrics described in section 6.3 — we found that this normalization yields greater numerical stability and improved results. However, for PPR, this improvement is marginal, and thus we opted not to include it for consistency and clarity. For HKPR, in contrast, the normalization had a substantial impact, reason why this caveat is discussed here.

Once again, we propose using $\psi = \#Visited_{\mathcal{Q}}^{TRAIN}$. The parameter t is a tunable scalar representing time. Intuitively, t controls the extent to which heat spreads from the initially selected nodes $UVis_{\mathcal{Q}}^{TRAIN}$ to the rest of the graph. At $t = 0$, HKPR returns ψ. As t increases, the values diffuse across the network according to Equation 6.4. Higher values of t result in a more uniform distribution, whereas lower values accentuate the local structure around the starting nodes. Thus, small t values emphasize local neighborhoods, while larger t promote global connectivity patterns. The Laplacian L governs how the initial heat distribution ψ evolves over time t. Notably, HKPR lacks a recursive formulation, in contrast with PPR. Employing the symmetrically normalized Laplacian ensures the diffusion process is degree-balanced, as the transfer of "heat" (score) from neighboring nodes depends on both their out-degrees and that of the node under consideration.

HKPR offers several advantages over PPR:

1. **Diffusion Weighting.** In Equation 6.3 and Equation 6.4, k denotes the length of the random walk. While PPR weights walks of length k by $(1-\alpha)^k$ (a geometric progression), HKPR weights them by $\frac{t^k}{k!}$ (an exponential progression). Consequently, node importance decays much faster with respect to the number of hops in HKPR compared to PPR. As demonstrated in section 6.3, this yields superior empirical performance.

2. **Degree Bias.** Because of its geometric aggregation, PPR tends to concentrate a substantial part of the probability mass in high-degree nodes. In contrast, HKPR maintains more probability mass within tightly connected communities, thereby better capturing graph structure. Indeed, HKPR is more effective in identifying low-conductance sets than PPR [30], making it preferable for clustering as it highlights nodes within well-separated communities and bridge nodes on sparse cuts.

3. **Empirical Performance.** As shown in section 6.3, HKPR consistently outperforms other methods across the majority of evaluation metrics and experimental settings.

In addition, HKPR inherits all the advantages of PPR described in subsection 6.2.4, compared to previous methods.

As with PPR, a variety of implementation strategies exist, each with distinct trade-offs. One algorithm for heat kernel computation [73] achieves a time complexity of $\mathcal{O}(t\frac{\ln(|V|/p_f)}{e_r^2\delta})$, where e_r is the relative error tolerance, δ is the threshold for significant normalized HKPR values and p_f is the allowable failure probability, and memory complexity of $\mathcal{O}(|\mathcal{E}|+|\mathcal{V}|+t\frac{\ln(|V|/p_f)}{e_r^2\delta})$ (where $|\mathcal{E}|+|\mathcal{V}|$ is the size needed to store the graph). In practice, however, HKPR may be slower than PPR on large-scale graphs, though smaller values of t typically accelerate convergence.

Despite these advantages, HKPR shares certain drawbacks with PPR. In particular, the introduction of the additional hyperparameter t increases the complexity of model selection, adding an extra variable to optimize. Its computational cost is also greater than the baseline, EVS and Expand from the Visited Set In-Degree (EVSI). Furthermore, compared to PPR, the interpretability of HKPR is reduced, making it more challenging to reason about its outputs and inner working. Lastly, just like PPR, there are more methods to approximate these scores that are specifically tailored for large graphs, scaling better as graph size increases [9, 73].

6.2.6 Other Methods

Several additional methods were explored during this work but are not included in the main results, as their performance fell short of expectations. For completeness, they are briefly mentioned here along with the rationale for their exclusion.

Firstly, all aforementioned algorithms were also evaluated using the number of times a node appeared on the search path, rather than the input array $\#Visited_Q^{TRAIN}$. Nodes appearing

on the search path constitute a subset of those visited, as defined in section 3.3. Across all methods, this alternative input led to inferior performance relative to using $\#Visited_{\mathcal{Q}}^{TRAIN}$. The likely cause is a weaker learning signal, as $\#Visited_{\mathcal{Q}}^{TRAIN}$ more closely aligns with the primary objective: ensuring that all vectors required for distance computations during search are available in memory. This set is precisely $visited(q)$, with q denoting a query.

Additionally, a variant of EVS was considered, in which the expansion started from the set of all unique nodes ever returned as results, i.e., $\bigcup_{q \in T} topk(q)$, where T is the set of training queries. The intuition was that queries from similar workloads would have overlapping *top-K* results. However, for small Θ values, while most elements in B are used, most queries have only a minimal overlap with the nodes in memory. This approach did not meet initial performance expectations and was therefore abandoned.

Finally, we evaluated the *Katz centrality* [28], a centrality measure that aggregates the centrality of a node's neighbors. Katz centrality has two parameters: α (a scalar) and β (a personalization vector akin to ψ). However, α must satisfy $\alpha < \lambda_{max}^{-1}$, where λ_{max}^{-1} is the largest eigenvalue of the graph's adjacency matrix. This requirement complicates parameter selection, needing either explicit computation of all eigenvalues or overly conservative setting of α. Moreover, Katz centrality exhibited unstable empirical performance, performing decently in some configurations but performing poorly in others. For these reasons, we omitted it from further experimentation.

6.3 Experiments

In this section, we test our methods, using different workloads and metrics. Following some of the findings, we introduce new research questions and their respective answers and metrics used.

6.3.1 System Architecture

The system consists of two primary components: the `Python` process - denoted as process A — and the process (or set of processes) within the VDBMS that receives and executes queries from process A, denoted as process B.

Firstly, process A requests process B to dump the HNSW layer 0 graph to a file, which is then loaded as a `csr_matrix` using `Scipy`. For each training query, process A sends a k-ANN search request to process B. Process B executes the query and returns the results to process A, which records the query recall, the search path, all visited nodes, and the top-1 result.

From the set of all nodes visited across all training queries, process A constructs a *visited frequency array*, indicating for each node the number of visits during the training phase. Using

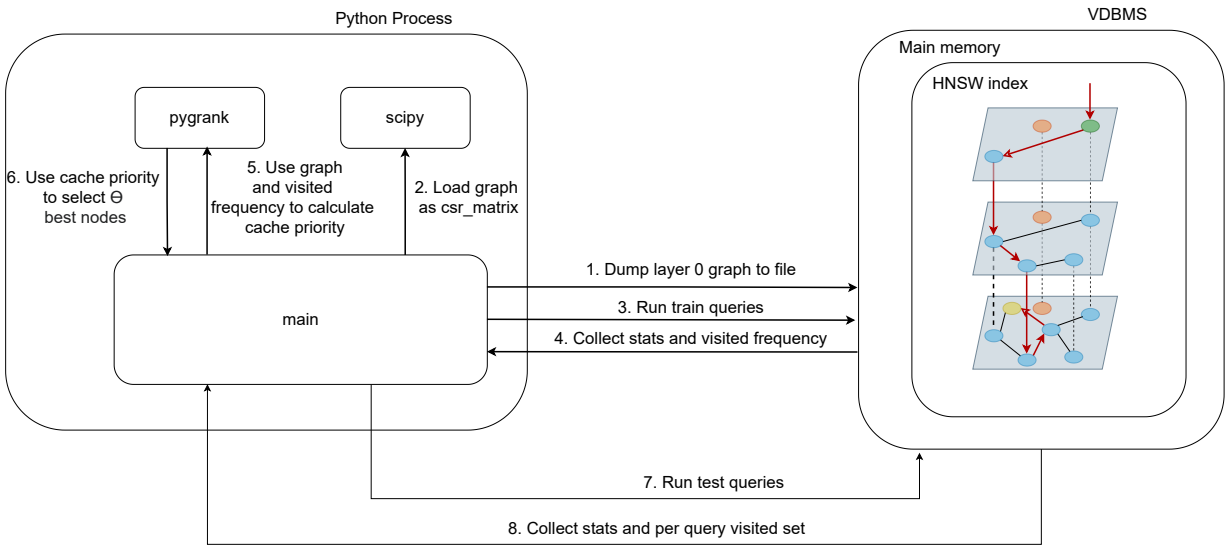


Figure 6.5: System architecture used to run experiments concerning vector selection for caching

this array and the graph representation, process A computes the cache priority policy, either via custom Python code with `NumPy` (for EVS, EVSI, and the baseline) or via `Pygrank` (for HKPR and PPR). Based on the selected cache priority policy and the *memory budget* constraint, process A determines the set of nodes B to retain in memory, choosing those with the highest priority.

For each test query, process A repeats the process: issuing a k-ANN search request to process B, which executes the query and returns results. Process A collects the same statistics as in the training phase. These results are then used to compute the evaluation metrics discussed in this section.

Some experiments adopt slightly modified configurations. For example, in subsection 6.3.7, step 4 is adapted so that nodes outside B are ignored during the search. However, other setups are only minor variations on this configuration and can be inferred from this section in conjunction with the experiment-specific explanations provided in the corresponding subsections.

6.3.2 Experimental Setup

We evaluate our methods using the datasets *SIFT*, *SIFT10M*, *SIFT50M*, *GIST*, and *GloVe100* [25, 48, 51]. The *SIFT*, *SIFT10M*, *SIFT50M*, and *GIST* datasets are available at `http://corpus-texmex.irisa.fr`, whereas *GloVe100* can be obtained from `https://nlp.stanford.edu/projects/glove/`.

As outlined in chapter 4, the entire query sets could not be used directly, as they do not reflect a clustered query workload. Instead, we extracted a representative workload using the procedure described in chapter 4. This method has two tunable parameters: the number of clusters C

and the number of nearest query vectors to each seed, L. The specific values for C and L are provided for each experiment.

Unless otherwise stated, the selected queries are split as follows: 50% are used to determine the nodes to retain in memory (i.e., generating B for each method) - the train queries - and the remaining 50% are used exclusively for evaluation, enabling direct comparison between the sets B produced by different methods - the test queries. The train-test split is random.

Each split is executed against the HNSW index, which is instrumented to track visitation statistics. For each query q, the index records $visited(q)$. This data is aggregated across the training queries to build $UVis_{\mathcal{Q}}^{TRAIN}$ and $\#Visited_{\mathcal{Q}}^{TRAIN}$. For test queries, we maintain the set $\{visited(q) \mid q \in T\}$, where T denotes the test set, to measure the proportion of each query's visited nodes that reside in memory.

HNSW indexes are built using *ef_Construction* = 300 and *M* = 32. Exploring how our methods perform across different building parameters is left as future work. Other parameters, such as *top-K* and *ef_Search*, are set per experiment. All methods are implemented and executed using *Python* 3.13.2. The HNSW layer 0 graph is imported as a `scipy.sparse.csr_matrix` using `Scipy` [69] v1.15.3. The Baseline (MFU), EVS, and EVSI are custom implementations, while PPR and HKPR are implemented via the `pygrank` [31] library (v0.2.14) with `NumPy` [19] v2.2.6 as the computational backend. For PPR (resp. HKPR), we apply `pygrank`'s *PageRank* (resp. *HeatKernel*) filter, using the personalization vector defined in subsection 6.2.4 (resp. subsection 6.2.5). The α (resp. t) parameter is set per experiment, and both methods are executed with a tolerance of $1\text{E}{-12}$.

6.3.3 Average In-Memory Set Hit Rate

The first metric used for evaluating our methods is the average in-memory set hit rate, which quantifies the proportion of nodes in $visited(q)$ that are present in the memory-resident node set B, expressed as a percentage and averaged over all test queries. Formally, this metric is defined as

$$\frac{\sum_{q \in T} |visited(q) \cap B|}{|T|} \times 100,$$

where T denotes the set of test queries.

A value of 100% indicates that all nodes visited by all test queries reside fully in memory, enabling queries to be answered without any disk access, even though only a subset of vectors are cached. We analyze this metric while varying *ef_Search* and *memory budget*. The *memory budget* (or Θ) is presented as a percentage of the total dataset size, allowing for comparisons among datasets.

The evolution of the metric with respect to increasing memory budget *memory budget* (Θ) is illustrated in Figure 6.6, while Figure 6.7 shows the metric variation over different *ef_Search*

values, keeping *memory budget* fixed. The experimental parameters for each subplot are detailed in the respective figure legends. Notably, the *memory budget* used varies for each dataset in Figure 6.7, reflecting differences in dataset characteristics discussed in subsection 6.3.4. We provide our rationale behind the choice of PPR parameter α (resp. parameter t for HKPR) in subsection 6.3.11.

Each barplot corresponds to a dataset, with colors representing methods introduced in section 6.2. As expected, the average set hit rate increases with higher *memory budget* due to more vectors being cached, enhancing the likelihood that queried nodes reside in memory.

Regarding *ef_Search*, two cases emerge from Figure 6.7:

1. For datasets like *SIFT*, *SIFT10M* and *SIFT50M*, the average hit rate is already high for low *ef_Search*; thus, methods except HKPR and PPR tend to degrade as *ef_Search* increases. This is because while *memory budget* is constant, the number of visited vectors grows with *ef_Search*, reducing the cache hit probability.

2. For other datasets where the average hit rate is not saturated at low *ef_Search*, performance improves with increasing *ef_Search*. This is partially explained by a corresponding rise in train-test overlap of visited nodes, as shown in Figure 6.11. Since the baseline, EVS, and EVSI methods rely heavily on train query data and have poor generalization capabilities, higher overlap benefits these methods despite increased number of unique visited vectors.

Conversely, PPR and HKPR demonstrate minor performance variation across *ef_Search* values due to their inherent generalization capabilities.

When $\Theta \leq |UVis_{\mathcal{Q}}^{TRAIN}|$ for given dataset, *ef_Search*, and *memory budget* combinations, baseline, EVS, and EVSI yield identical results, as all choose the same node set B. However, beyond this threshold, EVS and EVSI outperform the baseline by avoiding random guesses in node selection, with EVSI showing superior performance by prioritizing nodes with higher in-degree (which has been shown to correlate with visit likelihood [44]) out of all nodes at the same distance from the closest node in $UVis_{\mathcal{Q}}^{TRAIN}$.

Overall, PPR and especially HKPR consistently outperform other methods across datasets, *memory budget*, and *ef_Search* values. The exponential decrease in influence of a node in the score of other nodes, with respect with the number of hops from HKPR - contrasting with a geometric decrease from PPR, which is not as aggressive - aligns well with the problem, yielding the best results. At high *memory budget* levels, however, EVS and EVSI tend to match the results of the random-walk methods, reaching a saturation point.

While all proposed methods match or exceed baseline performance, the differences remain modest. Crucially, this metric alone does not reveal the distribution of hit rates across queries. For instance, an average hit rate of 95% might reflect most queries having 100% hit rate, with a minority having low hit rates, or it might reflect all queries having a 95% cache hit rate.

This distinction is important as maximizing the number of queries that do not need any disk I/O is the ultimate goal. Given disk access latency compared to memory, even one I/O operation can significantly degrade query performance. Moreover, due to vector storage layout and techniques like *piggybacking*, the difference in latency between fetching one or several vectors from disk is often negligible. These considerations are explored further in subsection 6.3.5.

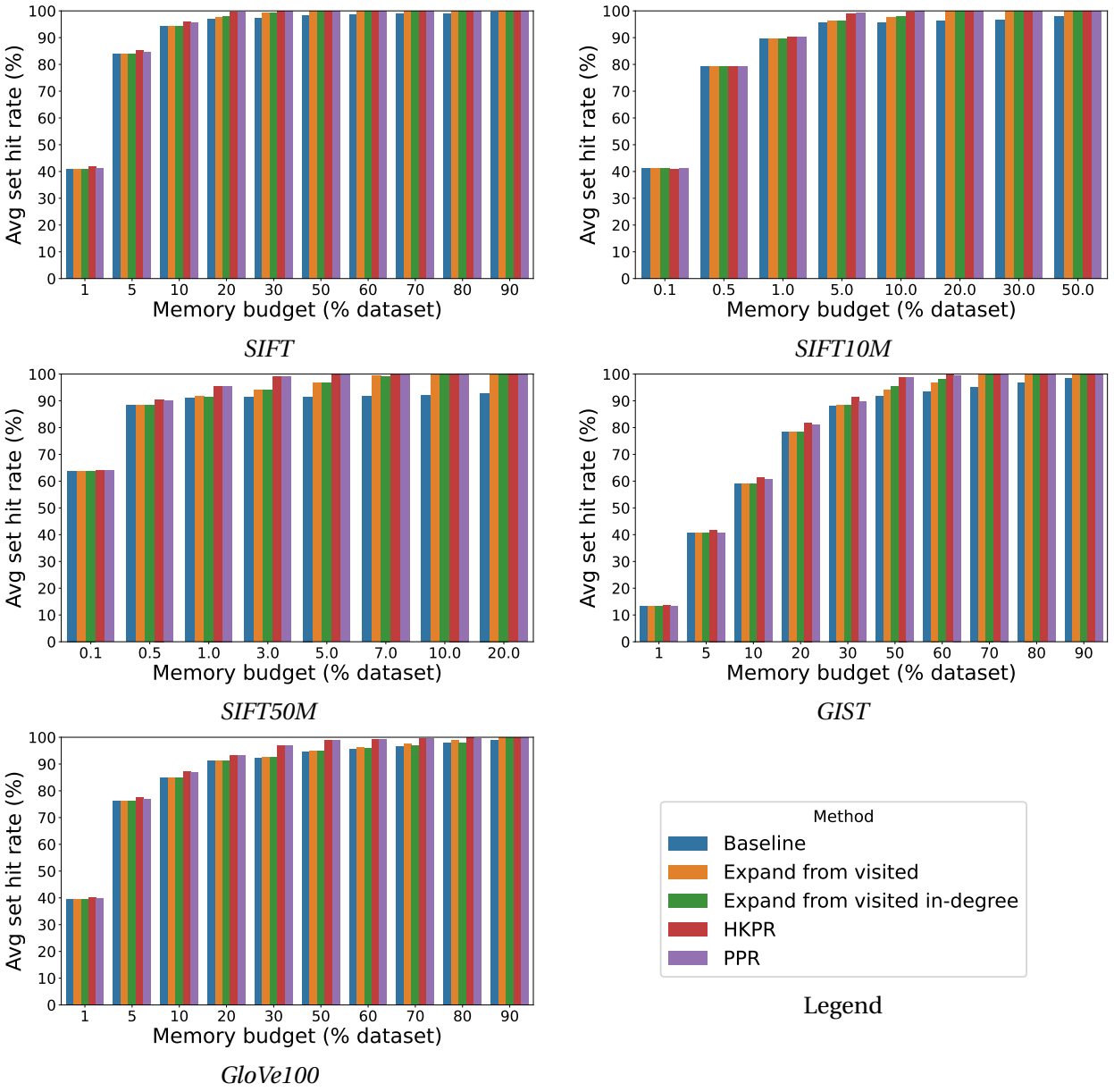


Figure 6.6: Barplot of average in-memory set hit rate over *memory budget*. For workload: $L = 300$ and $C = 1$. *top-K* = 10, *ef_Search* = 256, PPR $\alpha = 0.05$, HKPR t = 2.

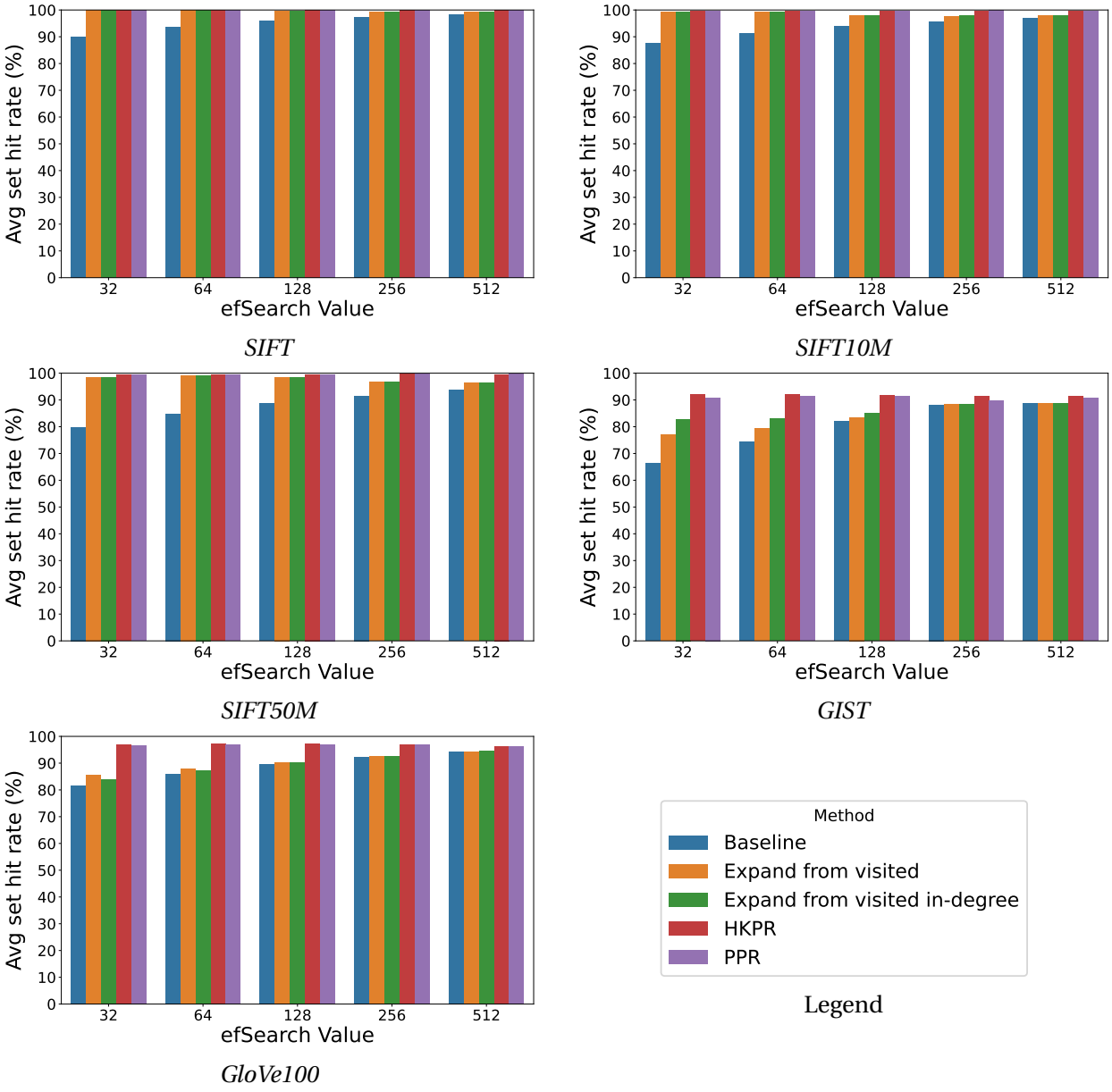


Figure 6.7: Barplot of average in-memory set hit rate over *ef_Search*. For workload: $L = 300$ and $C = 1$. *top-K* = 10, PPR $alpha = 0.05$, HKPR t = 2. For *SIFT10M*, *memory budget* = 10%, for *SIFT50M* *memory budget* = 5%, while for the rest *memory budget* = 30%

6.3.4 Percentage of the dataset visited and train-test overlap

Understanding the fraction of dataset nodes visited by train and test queries during HNSW search, as well as the overlap between these sets, is essential for interpreting results. These factors influence the minimum *memory budget* required to achieve high cache hit rates.

For example, if test queries visit 10% of the dataset nodes, then setting *memory budget* below 10% would never allow all queries to have all of the accessed nodes in memory and thus not necessitate any disk I/O.

Critically, for the same workload, the number of visited vectors does not scale linearly with dataset size, as illustrated in Figure 6.8. **This implies that as datasets grow, a smaller relative *memory budget* suffices to maintain performance, motivating our use of lower *memory budget* for *SIFT10M* and *SIFT50M*.**

Conversely, if vectors visited by train queries differ significantly from those visited by test queries, obtaining good generalization and performance is challenging. Methods like baseline, EVS, and EVSI - which have limited generalization capacity - are most impacted.

In this subsection in particular, the blue color represents generic data, not any method.

As observed in Figure 6.9 and Figure 6.10, the count of unique nodes visited varies substantially even with consistent query count ($L = 300$). However, node visits invariably increase with *ef_Search* due to greater exploration of the search graph.

The peculiarities of each dataset reveal further insights:

- *GIST*'s smaller test set (1000 queries) results in query vectors that are more dispersed, causing visits to diverse graph regions and more unique nodes.
- *SIFT10M*'s and *SIFT50M*'s larger dataset size means that although the number of nodes visited per query is greater, the percentage of total nodes visited is smaller, reflecting sub-linear scaling of visits with dataset size.
- Despite similar sizes (1M and 1.18M, respectively), *SIFT* and *GloVe100* differ in visit percentages, possibly due to inherent differences in difficulty of the *greedy* search to converge or spatial distribution of query neighbors.

Figure 6.11 depicts the percentage overlap of visited nodes between train and test queries defined as

$$\frac{|UVis_{\mathcal{Q}}^{TRAIN} \cap UVis_{\mathcal{Q}}^{TEST}|}{|UVis_{\mathcal{Q}}^{TRAIN}|} \cdot 100.$$

Higher overlap facilitates better node selection for B, especially benefiting the baseline, EVS, and EVSI methods due to their poor generalization ability and reliance on train query visits. As *ef_Search* increases, overlap rises since more nodes are visited by both train and test queries derived from the same workload, due to increased graph exploration. As a consequence, the likelihood of a node to be visited by both a train and a test query increases.

SIFT exhibits significantly greater overlap than *GloVe100*, correlating with the better relative performance observed on *SIFT*. Between *SIFT*, *SIFT10M* and *SIFT50M*, the overlap negatively

correlates with the dataset size, despite the workload being the same. This is expected as there are more vectors situated in between each query vector in the search space, meaning that each single vector is less likely to be visited overall.

This analysis underscores the importance of dataset characteristics and workload overlap in designing and selecting cache priority policies for efficient Approximate Nearest Neighbor search.

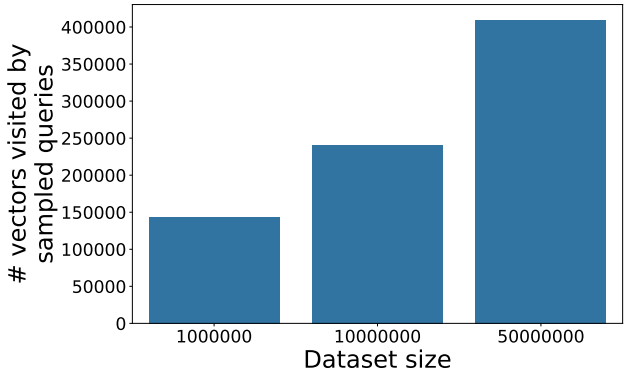


Figure 6.8: Number of unique vectors accessed by sampled queries, for *SIFT*, *SIFT10M* and *SIFT50M*. For workload: $L = 300$ and $C = 1$. *top-K* = 10 and *ef_Search* = 256.

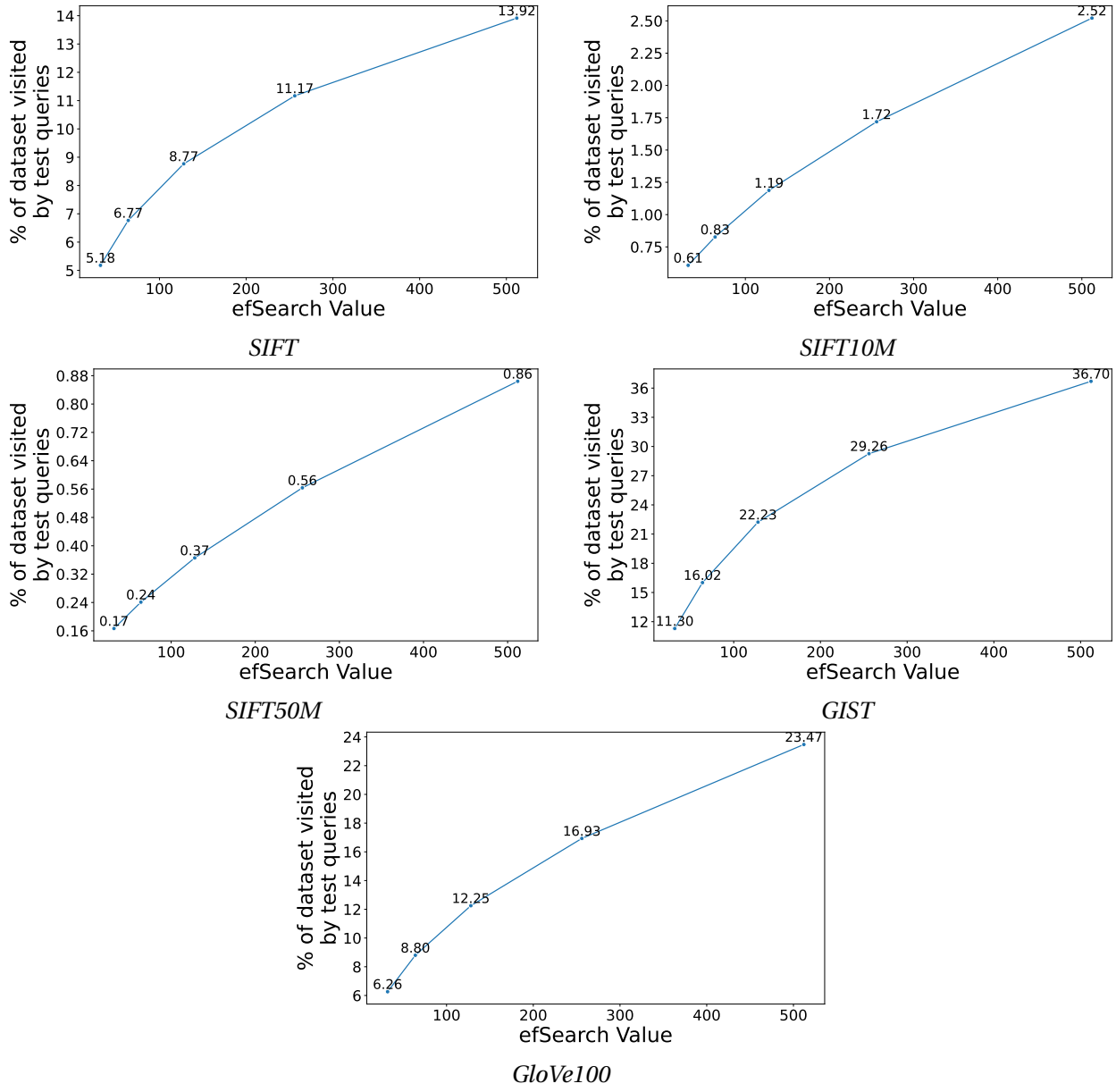


Figure 6.9: Lineplot of the percentage of the dataset visited by test queries, over *ef_Search*, during HNSW search. For workload: $L = 300$ and $C = 1$. *top-K* = 10.

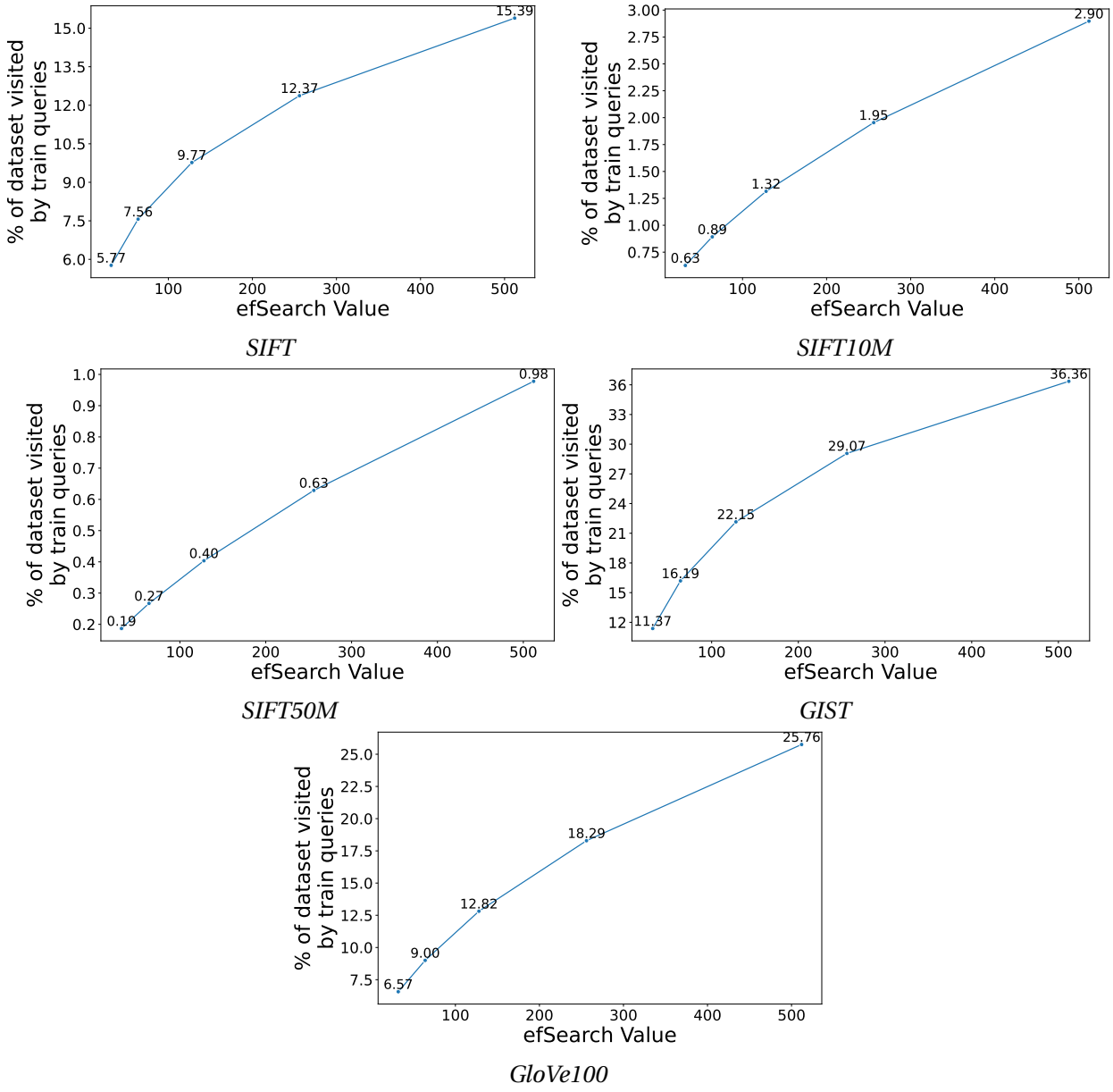


Figure 6.10: Lineplot of the percentage of total dataset visited by train queries, over *ef_Search*, during HNSW search. For workload: $L = 300$ and $C = 1$. *top-K* = 10.

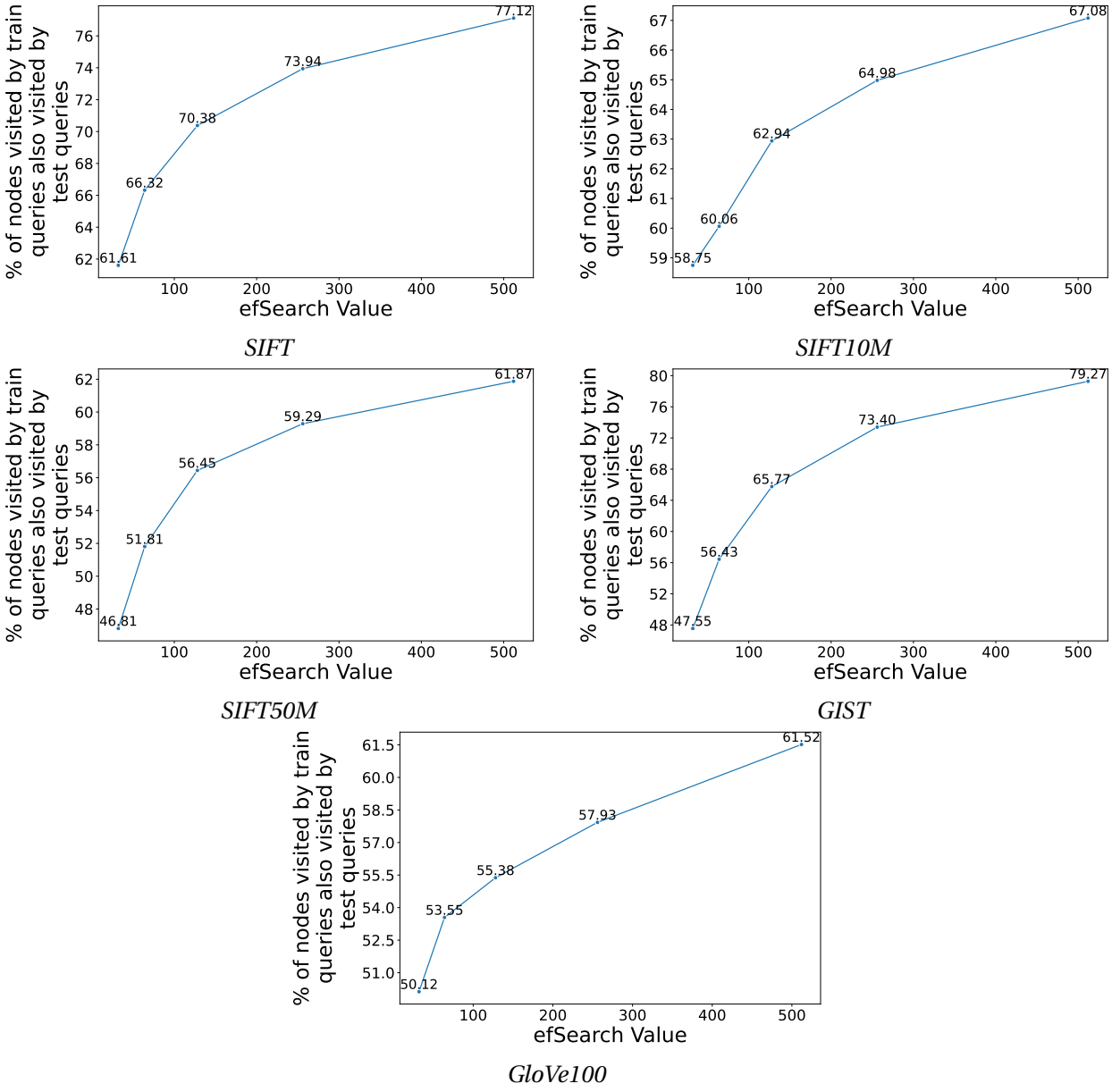


Figure 6.11: Lineplot of the intersection of nodes visited by train and test queries, over *ef_Search*. For workload: $L = 300$ and $C = 1$. *top-K* = 10.

6.3.5 Percentage of queries above overlap threshold

As noted in subsection 6.3.3, the average in-memory set hit rate alone is insufficient to fully evaluate the quality of each method. Since our primary objective is to maximize the number of queries that require no vector loading from disk, it is crucial to assess how many queries achieve this goal. To this end, we measure the percentage of test queries for which at least x% of the accessed vectors are contained in B, for each method.

Formally, this metric is defined as

$$m(x, B, T) = \frac{|X| \cdot 100}{|T|},$$

where T denotes the set of all test queries and

$$X = \Big\{ q \;\Big|\; x \leq |visited(q) \cap B| \cdot 100,\ \forall q \in T \Big\},$$

with $x \in [0, 100]$ representing the minimum percentage of accessed nodes in B required for a query to be included in X.

In Figure 6.12, the horizontal axis corresponds to different values of x, and the vertical axis plots $m(x, B, T)$ for each method (producing different B). Queries present in X for high x values require few or no I/O operations. Ideally, $X = T$ even for $x = 100$, meaning all queries can be answered entirely from memory.

As x increases, the performance gap between HKPR/PPR and other methods widens. The superiority of HKPR and PPR becomes particularly pronounced at x values close to 100, highlighting that **PPR—and especially HKPR—produce sets B that allow far more queries to be answered with minimal or no I/O**.

For example, in *SIFT*, with *ef_Search* = 256 and *memory budget* = 30%, **HKPR answers *100%* of queries with at least *99%* of accessed vectors in memory, compared to only about *73%* for EVS**. Furthermore, **HKPR answers about *82%* of queries with all vectors in memory, while EVS answers fewer than *8%***.

In *SIFT10M*, with *ef_Search* = 256 and *memory budget* = 10%, **HKPR answers approximately *95%* of queries with at least *99%* of accessed vectors in memory, whereas EVS answers fewer than *40%***. Figure 6.6 shows that for the same *memory budget*, the difference in average set hit rate is below 4%, yet the difference in the number of queries that can be answered with minimal to no I/Os is substantial.

In all datasets, methods based on random walks can serve many more queries entirely from memory compared to other approaches, in some cases achieving improvements of up to one order of magnitude. This is due to their ability to produce connected and traversable SG_B subgraphs, allowing queries to be resolved entirely from in-memory data. Additionally, as shown in subsection 6.3.8, when the set of training queries is significantly smaller than the set of testing queries, the performance gap between HKPR/PPR and the rest of the methods, widens, with the latter having their performance severely degraded.

A marked drop in $m(x, B, T)$ from $x = 99$ to $x = 100$ occurs across all datasets and methods. This is expected, as including every possible vector that similar queries might access is challenging. Since these methods do not explicitly use distances between vectors when determining what to store in memory, achieving perfect coverage within a reasonable execution time

is computationally impractical. Consequently, even with higher *memory budget* values, some datasets show no linear improvement in $m(100, B, T)$ (Figure 6.15). Based on this observation, in subsection 6.3.7, we investigate the effect on recall of entirely ignoring nodes on disk and force search to only use SG_B.

Figure 6.13 and Figure 6.14 depict the percentage of test queries with at least 99% of accessed nodes in memory as a function of *memory budget* and *ef_Search*, respectively. Both the used method and these parameters impact B and hence impact $m(99, B, T)$.

As expected, larger *memory budget* values generally lead to higher coverage. However, not all methods benefit equally: **HKPR and PPR serve substantially more queries at lower *memory budget* values than other methods.** For instance, in *SIFT10M*, with *memory budget* = 5%, HKPR serves over 80% of queries with at least 99% in-memory coverage, compared to around 25% for EVS. **To match HKPR's coverage, EVS requires a *memory budget* of around *20%*, four times larger.** Similarly, in *GIST* with *memory budget* = 50%, HKPR serves nearly 72% of queries, while EVS serves virtually none. Across all datasets, HKPR consistently outperforms PPR, albeit by a small margin, in handling more queries with minimal I/O at lower memory budgets. Moreover, comparing *SIFT*, *SIFT10M* and *SIFT50M*, the baseline's results scale worse with both *memory budget* and *ef_Search* as the dataset size increases, for the same workload. This is because, as seen in Figure 6.8, the percentage of the dataset visited by the same workload correlates negatively with the dataset size. Therefore, when $\Theta > |UVis_{\mathcal{Q}}^{TRAIN}|$, the baseline chooses $\Theta - |UVis_{\mathcal{Q}}^{TRAIN}|$ nodes randomly from an ever growing collection, relative to the *memory budget*.

From Figure 6.14, no consistent trend emerges: when near-optimal performance is achieved at low *ef_Search*, increasing *ef_Search* sometimes degrades results, as seen in *SIFT* with EVS. Conversely, when results are sub-optimal, increasing *ef_Search* improves performance for most methods except HKPR and PPR, as higher *ef_Search* increases overlap between training and test query access patterns (Figure 6.11).

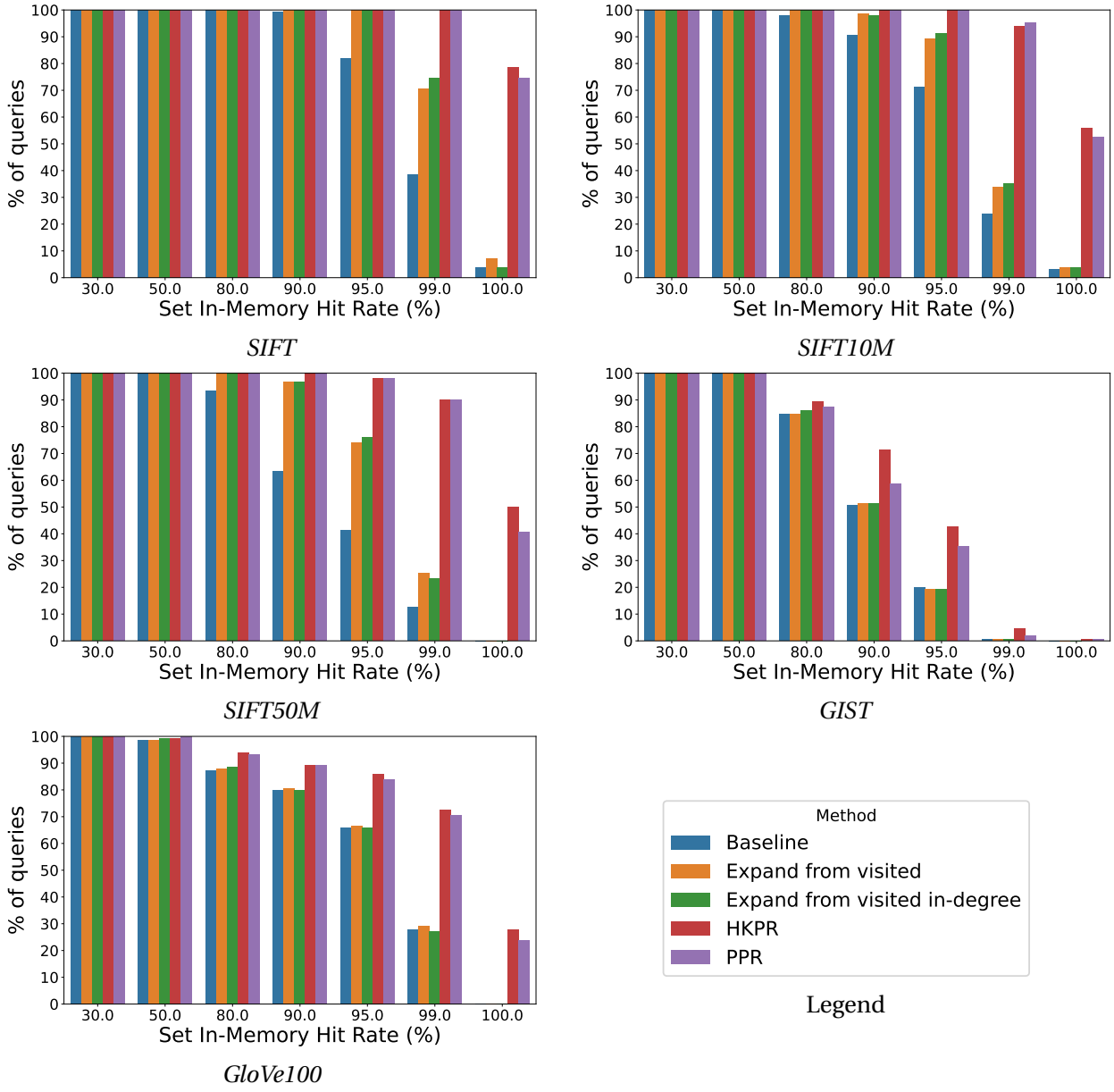


Figure 6.12: Barplot of the percentage of test queries (y-axis) over a certain set in-memory hit rate threshold (x-axis) . For workload: $L = 300$ and $C = 1$. *top-K* = 10, *ef_Search* = 256, PPR $alpha = 0.05$, HKPR t = 2. For *SIFT10M*, *memory budget* = 10%, for *SIFT50M*, *memory budget* = 5%, while for the rest *memory budget* = 30%

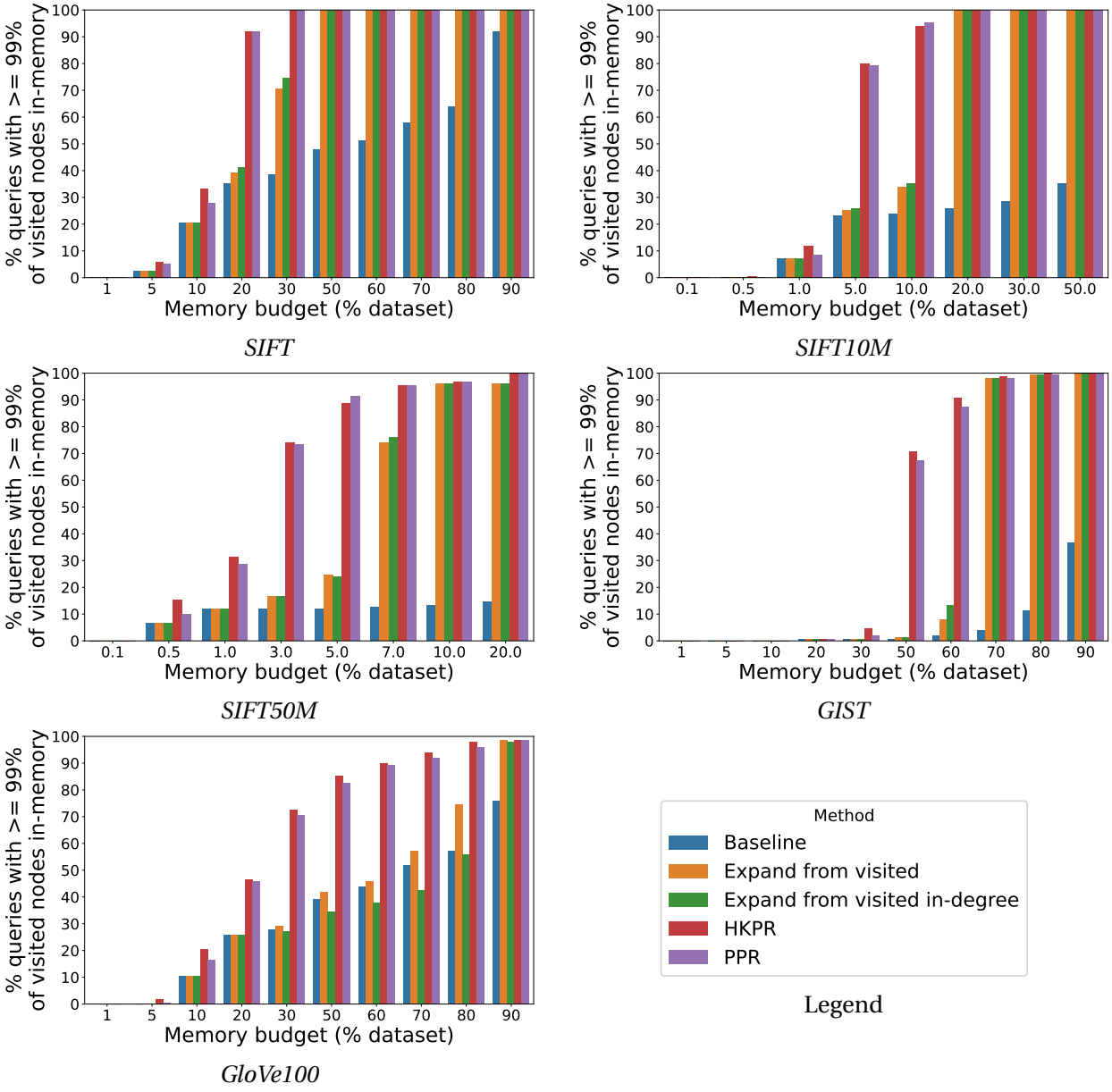


Figure 6.13: Barplot of the percentage of queries with at least 99% of the visited nodes in memory (y-axis) over *memory budget* (x-axis) . For workload: $L = 300$ and $C = 1$. *top-K* = 10, *ef_Search* = 256, PPR $alpha = 0.05$, HKPR t = 2.

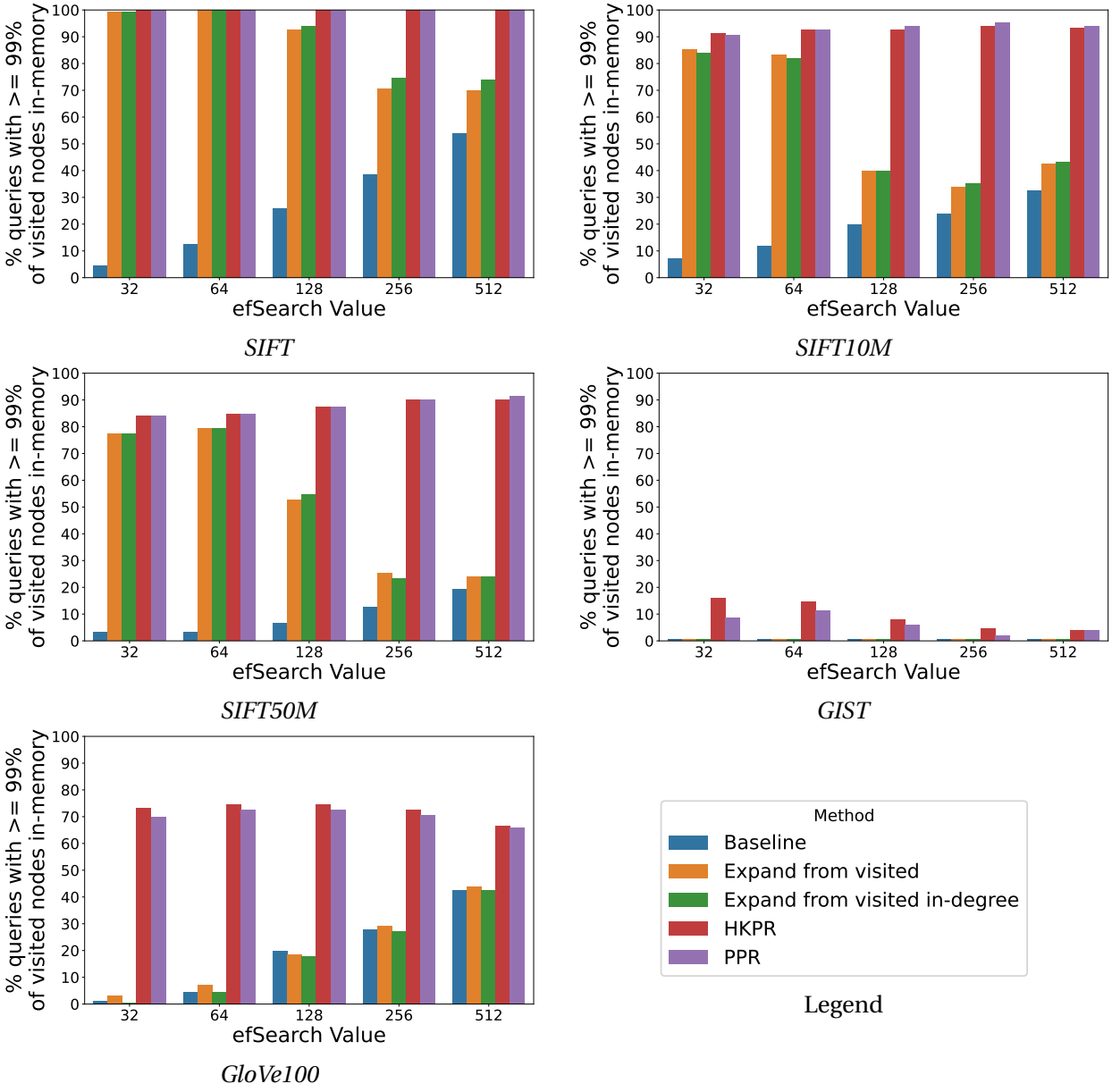


Figure 6.14: Barplot of the percentage of queries with at least 99% of the visited nodes in memory (y-axis) over *ef_Search* (x-axis) . For workload: $L = 300$ and $C = 1$. *top-K* = 10,PPR $alpha = 0.05$, HKPR t = 2. For *SIFT10M*, *memory budget* = 10%, for *SIFT50M*, *memory budget* = 5%, while for the rest *memory budget* = 30%

Figure 6.15 and Figure 6.16 present the percentage of test queries with *all* accessed nodes in memory over *memory budget* and *ef_Search*, respectively. Here, the performance advantage of HKPR over all other methods is even more pronounced. Achieving comparable results requires significantly higher *memory budget* values compared to the $x = 99\%$ case.

For example, in *GloVe100*, achieving at least 50% query coverage with full in-memory access

requires HKPR to store half the dataset, while EVS requires 90%. The baseline rarely exceeds 10% coverage, even with 90% of the dataset in memory, due to its random guessing strategy which fails to ensure a connected SG_B. Consequently, increasing *memory budget* produces minimal improvements for the baseline and for EVS/EVSI until a certain threshold is reached, after which results rise sharply (e.g., 50% for *SIFT*, 20% for *SIFT10M*, 90% for *GloVe100*, and 70% for *GIST*). Beyond these observations, the qualitative conclusions remain consistent with the earlier analys.

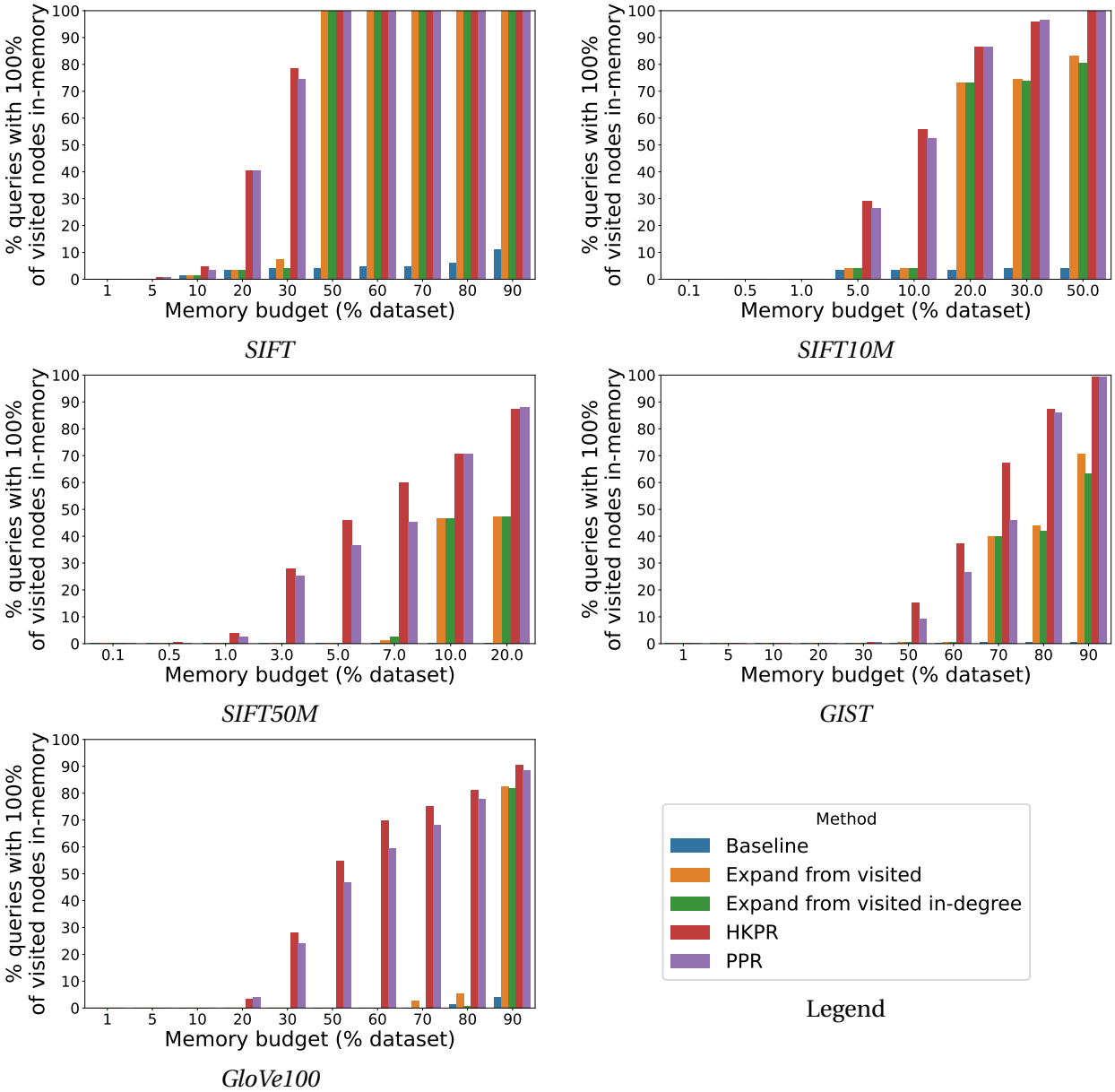


Figure 6.15: Barplot of the percentage of queries with all of the visited nodes in memory (y-axis) over *memory budget* (x-axis) . For workload: $L = 300$ and $C = 1$. *top-K* = 10, *ef_Search* = 256, PPR $alpha = 0.05$, HKPR t = 2.

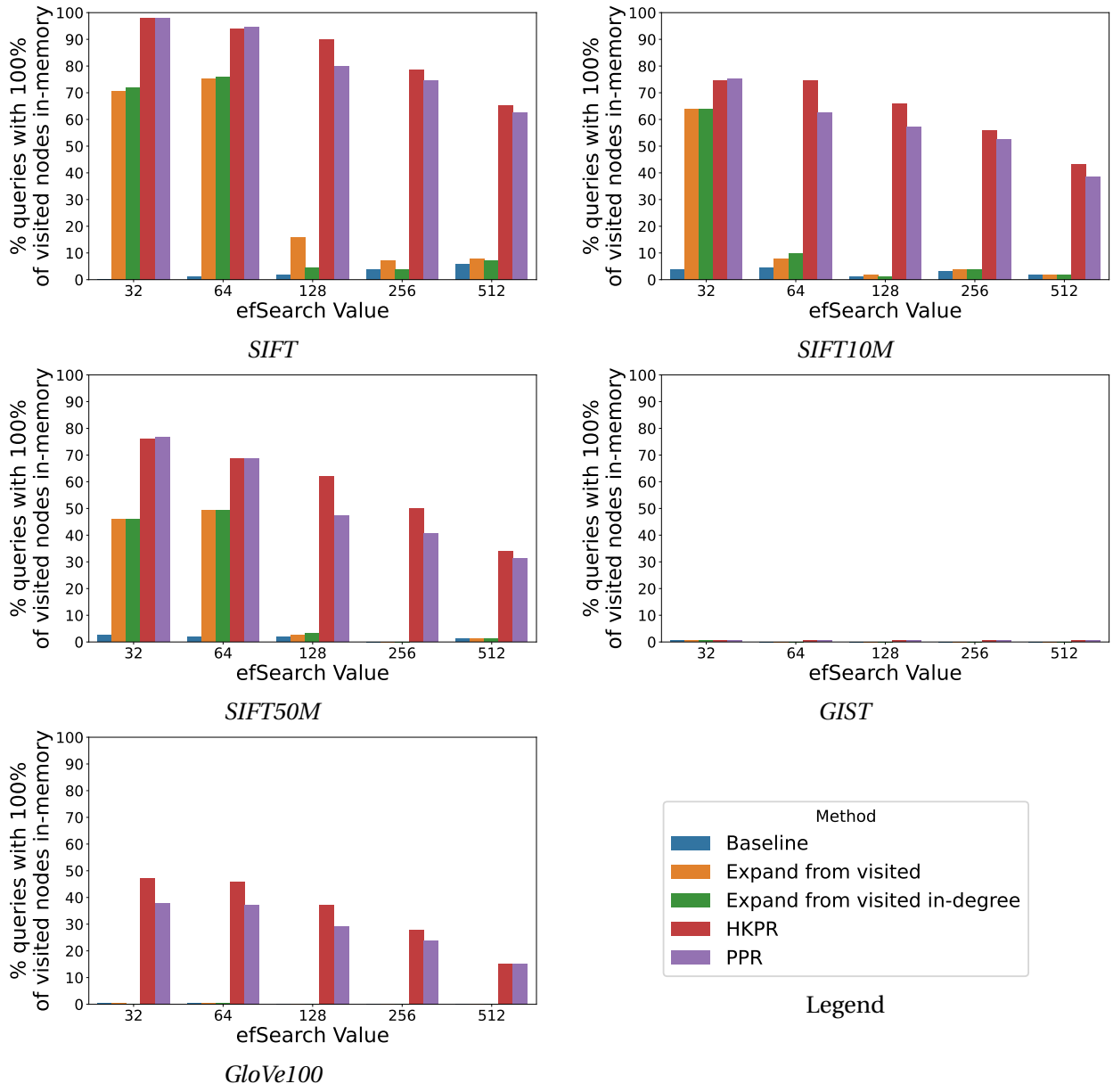


Figure 6.16: Barplot of the percentage of queries with all of the visited nodes in memory (y-axis) over *ef_Search* (x-axis) . For workload: $L = 300$ and $C = 1$. *top-K* = 10,PPR $alpha = 0.05$, HKPR t = 2. For *SIFT10M*, *memory budget* = 10%, for *SIFT50M*, *memory budget* = 5%, while for the rest *memory budget* = 30%

6.3.6 Scaling the number of queries

When using $L = 300$ and $C = 1$, the dataset consists of 150 queries for training and 150 queries for testing. Increasing L increases the number of available queries, but it also increases the cluster diameter. This occurs because queries are selected progressively farther from the seed

query, thereby expanding the hypersphere within which all nearby queries are located (chapter 4). Consequently, it is not feasible to scale the experiments to include more queries while keeping the cluster diameter constant.

One possible solution to this limitation is to artificially generate additional queries from the existing 300. For instance, one could fit a multivariate normal distribution to the available queries: the mean would represent the cluster centroid (expected to be close to the seed query, assuming a uniform distribution of nearby queries in all directions) while the covariance matrix would capture the cluster's spread across all dimensions.

However, we adopted an alternative approach. Several of the datasets provide an additional *learn split*, which can be used to train algorithms. This split contains a significantly larger number of vectors, enabling us to scale our experiments effectively. The *GloVe100* dataset does not include a learn split and is therefore excluded from this section. The *SIFT* and *GIST* datasets each include 100,000 learn vectors, *SIFT10M* contains 1 million learn vectors, and *SIFT50M* contains 5 million learn vectors. In all cases, the query-to-base set ratio is maintained at 1 query per 10 base vectors.

All metrics presented in previous experiments are equally applicable to the learning vectors. However, for conciseness, only the most relevant results are shown here. To maintain the same percentage of sampled queries (compared to the total set), we used $L = 3000$ for *GIST* and *SIFT*, $L = 30000$ for *SIFT10M* and $L = 150000$ for *SIFT50M*. This corresponds to sampling 3% of the *learn set* across all datasets. For all experiments, we set $C = 1$.

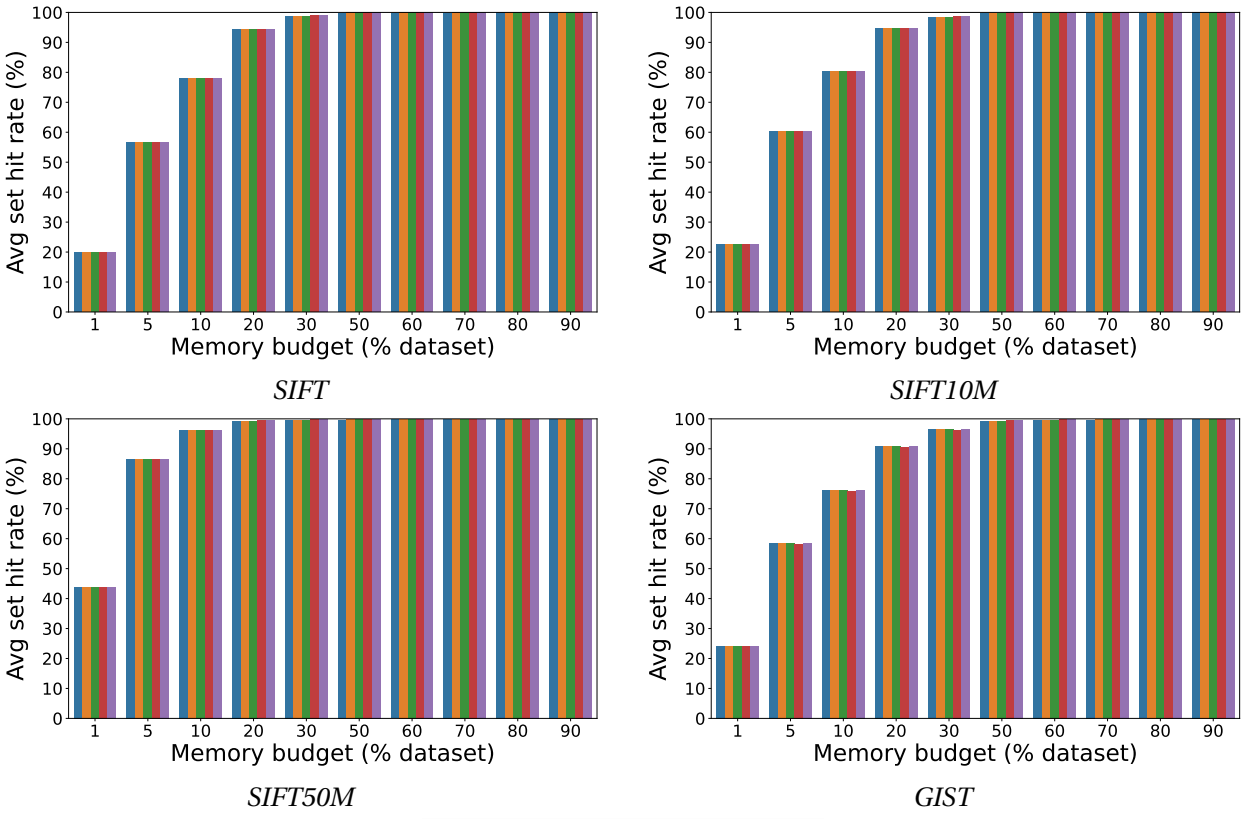
Figure 6.17 shows the average in-memory set hit rate for the *learn set* over *memory budget*. The first notable observation is that the performance differences between the evaluated methods are substantially smaller compared to subsection 6.3.3, which is unsurprising. In these experiments, 88.43% of vectors accessed by training queries are also accessed by test queries for *GIST*, with *SIFT*, *SIFT10M* and *SIFT50M* exhibiting values of 90.79% 87.83% and 84.14%, respectively. This high overlap particularly benefits the baseline, EVS and EVSI, as relying almost exclusively on training data with minimal generalization yields substantially improved results in this setting. The observed increase in overlap is also expected. Since the learn set is significantly bigger than the set used before, query density in the search space is also bigger, and the average vector distance between two randomly chosen queries decreases, resulting in a greater likelihood that their searches will visit the same nodes. Put differently, for a hypersphere of fixed radius around a seed query, a denser query set implies a substantially larger number of queries within that hypersphere.

Figure 6.18 presents the percentage of queries whose visited nodes are entirely stored in memory, for the *learn set*. This result is somewhat surprising: despite the negligible differences in average in-memory set hit rates across all methods, **HKPR and PPR still significantly outperform the others in terms of the proportion of queries that can be fully answered from memory.** This advantage stems from the same underlying factors discussed previously: both HKPR and PPR tend to form a connected SG_B, enabling complete traversals within the subgraph, and their

generalization capabilities are crucial for achieving a high percentage of fully in-memory query responses.

Although results for the percentage of queries with at least 99% of visited vectors in memory are omitted here for conciseness, the trends are consistent.

The key conclusion is that **even in scenarios where the overlap between nodes accessed by training and test queries is very high — significantly benefiting the baseline, EVS and EVSI — PPR, and especially HKPR, still deliver superior performance in terms of the percentage of queries served entirely from memory or with minimal I/O, owing to their generalization capacity and emphasis on subgraph connectivity**.



Legend

Figure 6.17: Barplot of average set hit rate over *memory budget*. For workload: $L = 3000$ for *GIST* and *SIFT*, $L = 30000$ for *SIFT10M*, $L = 150000$ for *SIFT50M* and $C = 1$. *top-K* = 10, *ef_Search* = 256, PPR $\alpha = 0.05$, HKPR t = 2.

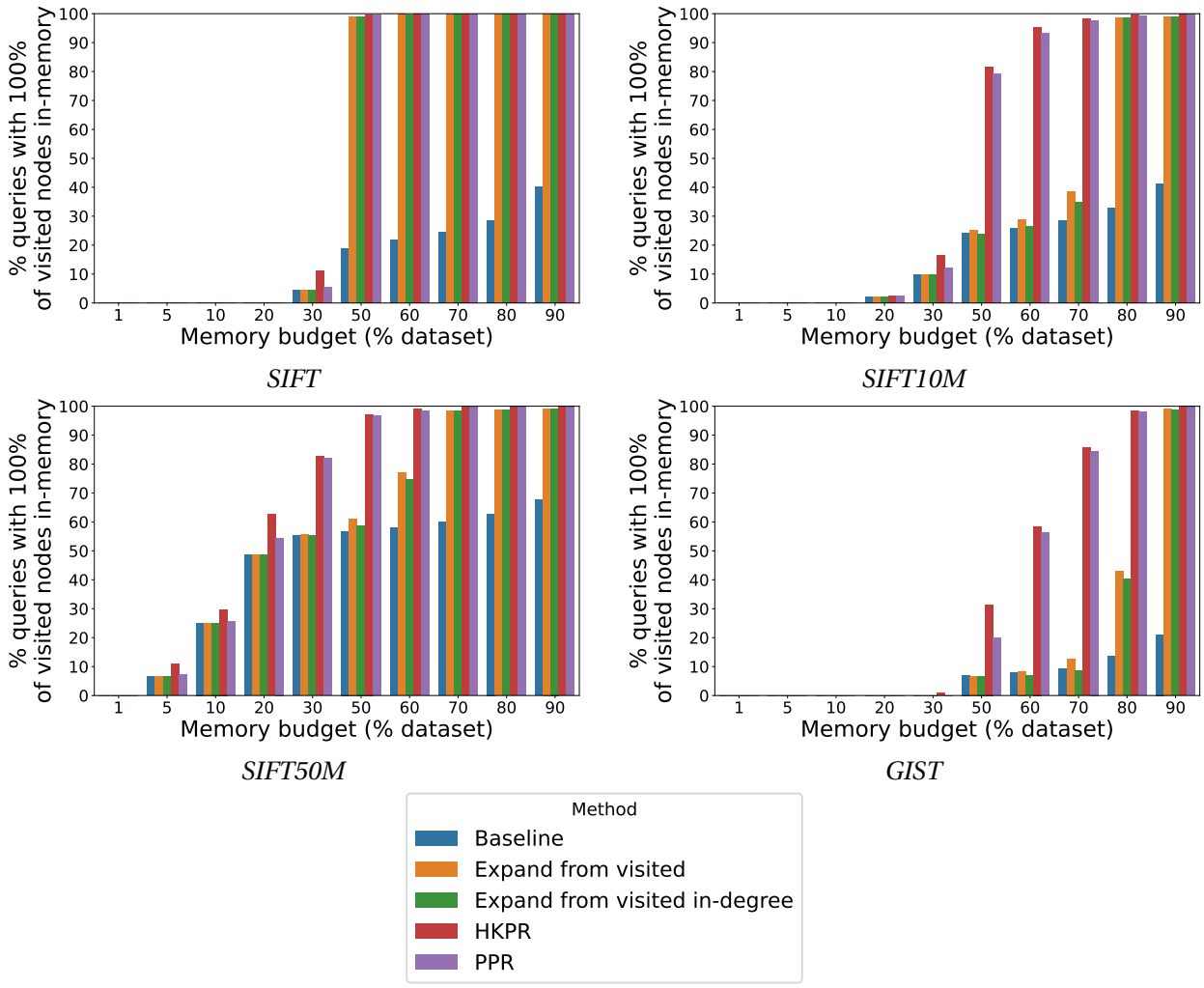


Figure 6.18: Barplot of the percentage of queries with all of the visited nodes in memory (y-axis) over *memory budget* (x-axis) . For workload: $L = 3000$ for *GIST* and *SIFT*, $L = 30000$ for *SIFT10M*, $L = 150000$ for *SIFT50M* and $C = 1$. *top-K* = 10, *ef_Search* = 256, PPR $alpha = 0.05$, HKPR t = 2.

6.3.7 Dismissing the vectors on disk

In this subsection, we investigate the effect of performing the search exclusively within SG_B, disregarding nodes not present in memory, rather than retrieving them from disk. If the resulting drop in recall is negligible, these nodes can be safely ignored, thereby enabling query processing with zero I/O operations, even when not all nodes are cached.

As seen in Figure 6.12, there is a considerable drop in the number of queries that have all of the visited nodes in memory compared to the number of queries that have at least 99% of the visited nodes in memory. If no uncached node can be ignored without a significant drop in

Recall, then only this much smaller percentage of queries can be answered fully from memory. Even though queries with at least 99% of the accessed nodes in memory have to fetch a small number of vectors from disk (1% or less of all accessed vectors), one single I/O can already significantly affect query latency, as accessing secondary storage is usually orders of magnitude slower than accessing main memory. Driven by these 2 observations, we explore whether there is a significant impact on Recall if the uncached nodes are disregarded for queries that already have at least 99% of the accessed nodes in memory. In case there is not, this much higher percentage of queries can be answered with about the same latency and recall of a fully in-memory HNSW index, while only keeping a fraction of the vectors in memory.

In multi-layered graphs such as HNSW, the entry point at layer 0 is dynamically chosen based on a combination of the upper layers and the query vector. Consequently, there may be cases where the selected entry point is not present in memory — when the corresponding upper layers are not cached — or where it has no neighbors stored in memory. In such scenarios, the search would terminate immediately, as the entry point would effectively be isolated.

To mitigate this situation, we implement a simple yet effective mechanism: if the entry point lacks any neighbors in memory, we instead select a random entry point in layer 0 that has at least one cached neighbor. Although this approach is straightforward, more sophisticated strategies could be considered. For instance, if the entry point in layer 0 has no neighbors in memory, one could perform a BFS in layer 1 starting from that node and halt upon finding a node with at least one neighbor in memory. This node would then serve as the entry point and would, with high probability, be closer to the query vector than a randomly selected candidate. Alternatively, the layer 1 search could be executed with a *top-K* parameter greater than 1 (in contrast to the standard search algorithm [37]). If the top-1 candidate has no neighbors in memory, the algorithm would try the top-2 candidate, followed by the top-3, and so on.

Nevertheless, our simple mechanism suffices to demonstrate our core argument: **in certain conditions, vectors not cached in memory can be safely ignored.** In Figure 6.19, Figure 6.20, and Figure 6.21, we present the average recall (across all test queries) for each method, when the search is restricted to SG_B, as a function of *memory budget*, *ef_Search*, and *top-K*, respectively. In these figures, an **additional column in brown represents the average recall of the fully in-memory standard HNSW search (as detailed in section 2.4)**.

From Figure 6.19, we observe that although HKPR achieves slightly higher recall than other methods, this advantage is marginal. Relative to standard search, and depending on the dataset, once *memory budget* exceeds a certain threshold, the recall difference between standard search and search restricted to SG_B becomes negligible. **Beyond this point, we argue that ignoring vectors stored on disk has minimal impact on recall, while enabling all queries to be answered without performing any I/O.**

Examining Figure 6.20, we find that increasing *ef_Search* reduces the differences between methods, with recall values converging toward those of standard search. In Figure 6.21, no significant recall improvement is observed with increasing *top-K*, except for minor variations in

specific datasets, which we consider statistically insignificant.

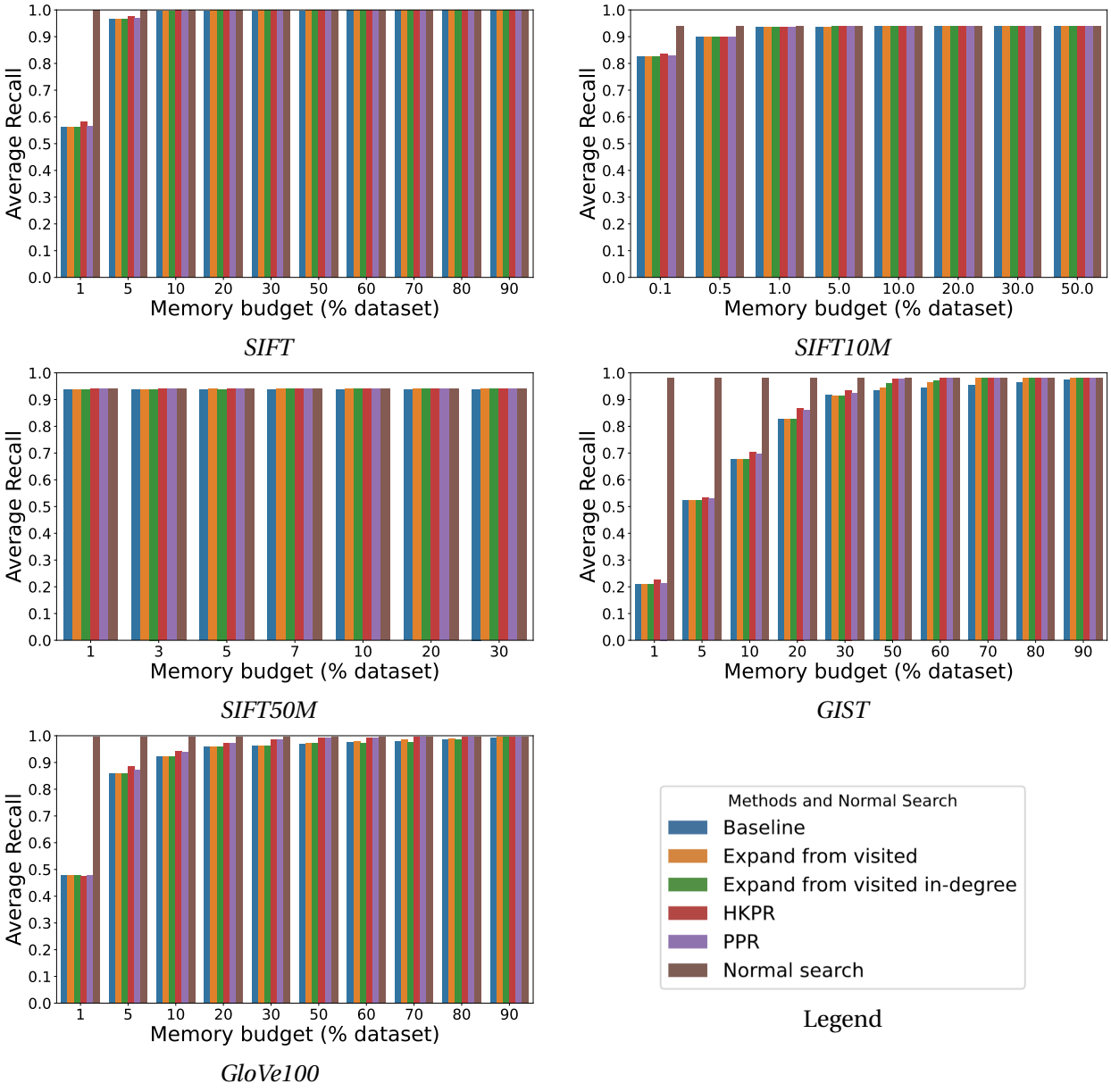


Figure 6.19: Barplot of average Recall@10 over *memory budget*. For workload: $L = 300$ and $C = 1$. *top-K* = 10, *ef_Search* = 256, PPR $\alpha = 0.05$, HKPR t = 2.

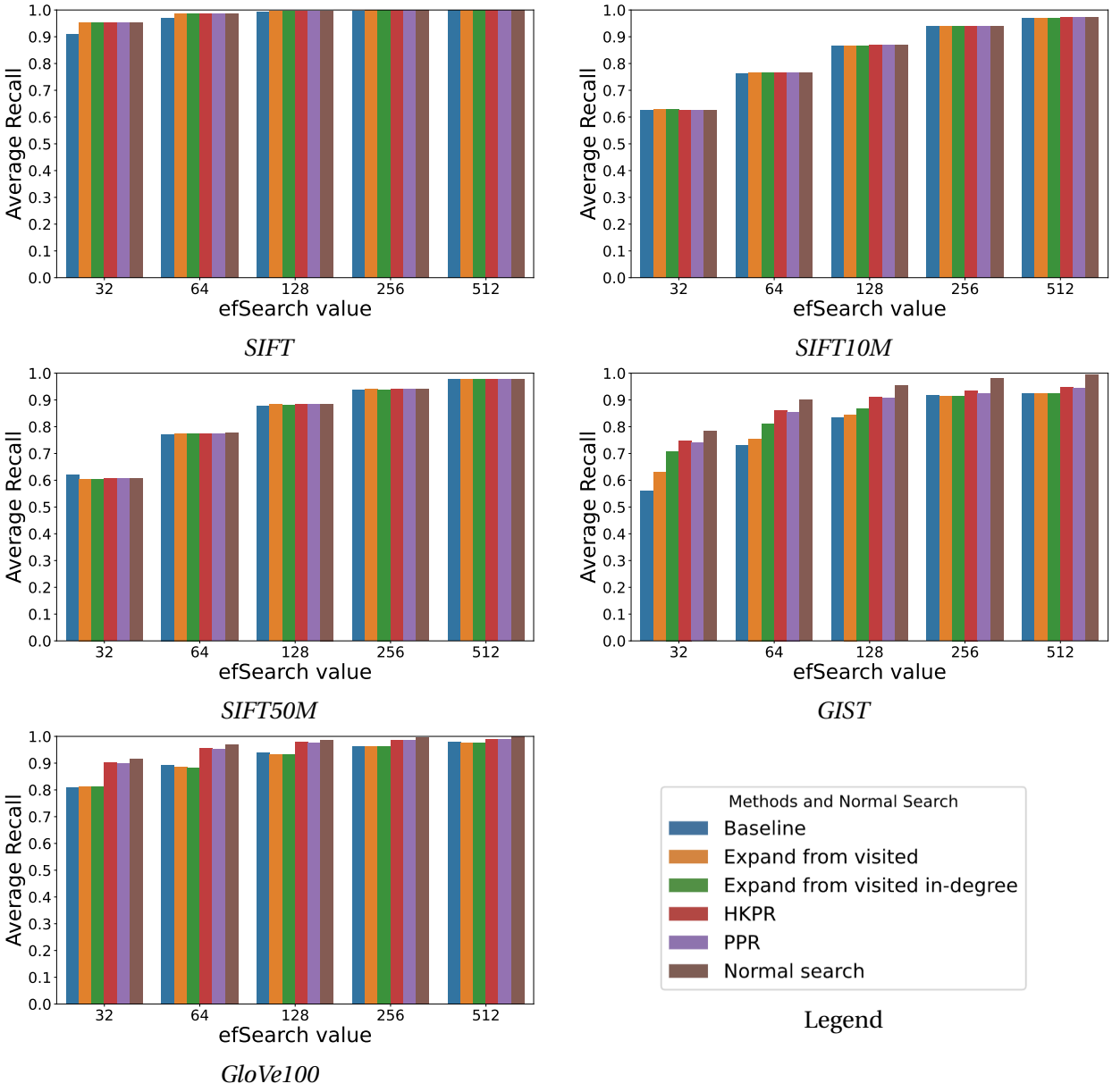


Figure 6.20: Barplot of average Recall@10 over *ef_Search*. For workload: $L = 300$ and $C = 1$. *top-K* = 10, PPR $alpha = 0.05$, HKPR t = 2. For *SIFT10M*, *memory budget* = 10%, for *SIFT50M* *memory budget* = 5%, while for the rest *memory budget* = 30%

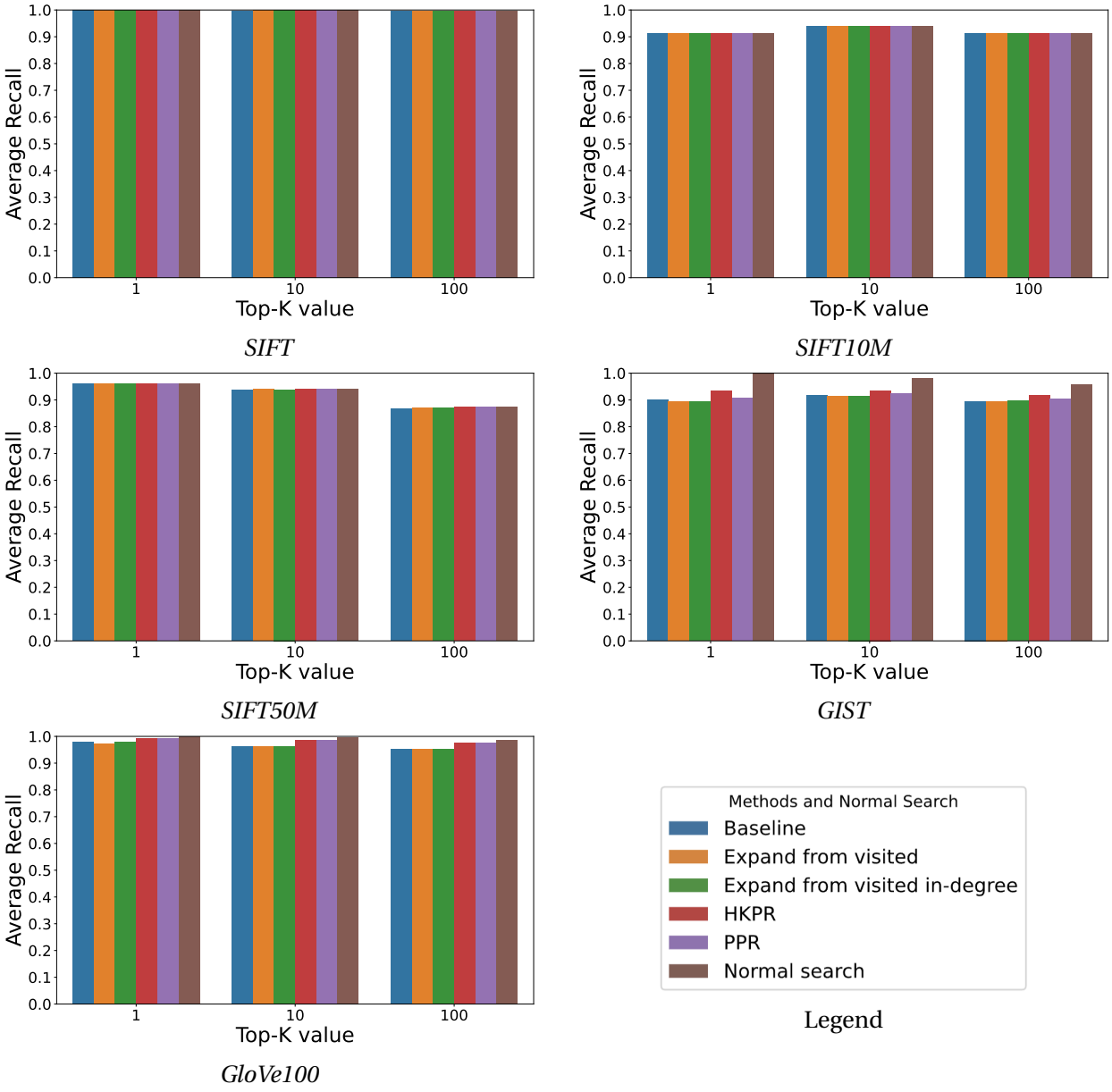


Figure 6.21: Barplot of average Recall@K over *top-K*. For workload: $L = 300$ and $C = 1$. *ef_Search* = 256, PPR $alpha = 0.05$, HKPR t = 2. For *SIFT10M*, *memory budget* = 10%, for *SIFT50M memory budget* = 5%, while for the rest *memory budget* = 30%

As noted in subsection 6.3.3, plots of the average recall across queries conceal important per-query differences. In Figure 6.22, we illustrate the average per-query recall loss between standard search and search restricted to SG_B for each method (y-axis), for queries with a certain percentage of accessed nodes in B during standard search (x-axis). Let $\mathcal{R}_N(q)$ denote the recall of a standard search for query q and $\mathcal{R}_B(q)$ the recall under search restricted to SG_B. The x-axis is divided into bins $]a,b]$ representing the percentage of accessed nodes in memory. For one such

bin, the y-axis represents:

$$\frac{\sum_{q \in X} \mathcal{R}_N(q) - \mathcal{R}_B(q)}{|X|}$$

where

$$X = \left\{ q \;\middle|\; a < 100 \cdot \frac{|visited(q) \cap B|}{|visited(q)|} \leq b, \;\; \forall q \in T \right\}$$

and T denotes the set of all test queries.

This analysis quantifies the recall impact of ignoring vectors on disk as a function of how many such vectors need to be discarded. Naturally, when all accessed nodes are in memory, recall loss is zero, since the execution path matches the standard search exactly. **Across all datasets, for queries where at least 95% of accessed nodes are in memory, the recall difference is negligible.**

For *SIFT* and *GIST*, we note that for queries with a small fraction of accessed nodes in memory, HKPR exhibits larger recall drops compared to the baseline, EVS, or EVSI. Two factors contribute to this:

- the number of queries assigned to a given bin vary across methods—HKPR may have only one query in $]30, 50]$, whereas the baseline might have many more — making the baseline's average drop in recall inherently smaller.
- HKPR captures the trends of the workload and is optimized to maximize the number of queries with nearly all vectors in memory, potentially sacrificing recall for rare outlier queries, whereas the baseline's random selection of nodes may better accommodate such cases.

Up to this point, we have considered a binary decision: either disregard all on-disk nodes for every query or always fetch them. This decision could instead be query dependent, using heuristics to decide whether to disregard a missing vector or not. For example, one could estimate the proportion of in-memory nodes within x hops of the entry point in layer 0: high proportions would classify the region as "hot" and permit in-memory only search, while low proportions would designate "cold" regions, requiring disk access. An alternative would be to start an in-memory search and, if too many nodes are ignored, switch to fetching vectors or restart with full disk access. Lastly, there could be heuristics that decide, for each time a vector needs to be fetched from disk, whether to disregard it or to fetch it, based on some importance ranking of nodes (e.g. in/out-degree, PPR/HKPR score, etc...). While such heuristics could balance recall and latency, we leave their design for future work.

Nevertheless, since executing searches restricted to SG_B for queries with at least 95% of accessed nodes in memory yields negligible recall loss — and given that HKPR achieves substantially more such queries than the baseline — ignoring on-disk vectors for these cases would allow a large fraction of queries to be answered with zero I/O, matching the latency of a

fully in-memory HNSW index and incurring only minimal recall degradation. The remaining queries, for which recall loss is non-negligible, would be executed with full recall but at the expense of higher latency. The threshold proportion of in-memory nodes beyond which disk access is skipped thus represents a key trade-off between latency and recall.

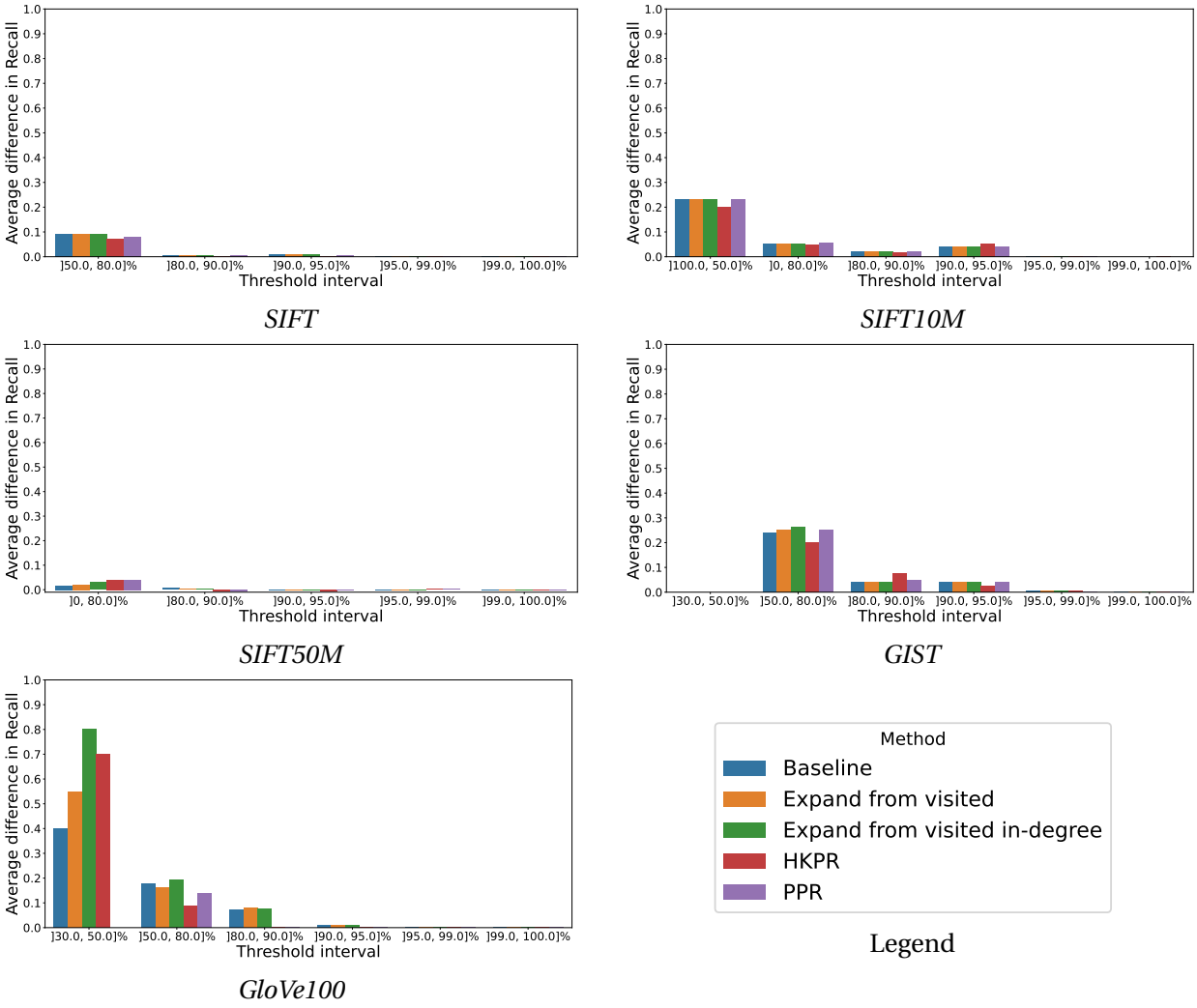


Figure 6.22: Barplot of average difference in Recall@10 for queries with different percentage of accessed nodes in memory. For workload: $L = 300$ and $C = 1$. *top-K* = 10, *ef_Search* = 256, PPR $alpha = 0.05$, HKPR t = 2. For *SIFT*, *memory budget* = 5%, for *SIFT50M* *memory budget* = 1%, for *SIFT10M*, *memory budget* = 0.5% while for the rest *memory budget* = 30%

6.3.8 Improvement over training signal

In this subsection, we evaluate the number of training queries required by each method to construct a high-quality set B for a fixed workload, and how sensitive each method is to the number of training queries. This experiment aims to closely mirror the practical decision-making

process of a real VDBMS, assuming our methods were applied in such a system. After processing a certain number of queries, the system must decide which subset of nodes should be retained in memory. The number of training queries used to determine this subset may critically affect both the quality of the selected set and the number of disk I/Os needed to serve subsequent queries. However, it is not clear in advance how many queries should be employed for this computation.

To address this uncertainty, as well as to understand what methods are more impacted by the training set size, we designed the following experiment. We employed the learn set (introduced in subsection 6.3.6) in order to obtain a sufficiently large collection of queries. We sampled 3% of the learn set as workload for each dataset, resulting in $L = 3000$ queries for *GIST* and *SIFT*, $L = 30000$ for *SIFT10M* and $L = 150000$ for *SIFT50M*. In all cases, we fixed $C = 1$. The test set remains constant throughout the experiments, defined as a uniform random sample of 70% of the workload. This yields 2100 test queries for *GIST*, *GloVe100*, and *SIFT*, 21000 test queries for *SIFT10M* and 105000 test queries for *SIFT50M*. From the remaining queries (normally used as training queries in other setups), we uniformly and randomly sampled a fraction for training.

The x-axis values in our plots correspond to the number of queries chosen for training under this second sampling procedure. Notably, while the test set is always the same and significantly larger than the training set — reflecting what would occur in a real system — our analysis focuses on determining how many training queries are required for different methods to achieve satisfactory results. Specifically, we experimented with sampling 1%, 5%, 10%, 20%, 30%, 50%, 70%, and 100% of the training set, which itself consists of 30% of all sampled queries.

Figure 6.23 presents the average in-memory set-hit rate as a function of the number of training queries. Once again, the performance difference between HKPR and PPR compared to other methods remains small, with the exception of the baseline. The baseline is highly sensitive to the size of the training data. When the training set is small, its performance degrades sharply relative to the workload. **This is an undesirable property for any method intended to solve this problem: requiring as many or more training queries than test queries in order to construct a high-quality cached set is impractical.** In real-world systems, the aim is to rely on a small training set while generalizing effectively to a much larger stream of incoming queries.

Figure 6.24 and Figure 6.25 depict, respectively, the proportion of queries with at least 99% of accessed nodes in memory and the proportion with complete coverage. Here, PPR and especially HKPR consistently outperform the alternatives. With as few as 45 queries for *SIFT* and *GIST*, or 450 queries for *SIFT10M*, HKPR achieves results only slightly inferior to those obtained with 900 or 9000 queries, respectively, thus demonstrating strong generalization capacity. Importantly, **HKPR achieves results with as few as 45 (resp. 450) training queries that cannot be matched by the baseline, EVS, or EVSI even with 900 (resp. 9000) queries.** This property, on the other hand, is highly desirable: provided that the sampled queries are reasonably representative of the workload, performance is nearly identical to that obtained with far larger training sets. This shows that HKPR and PPR are far less sensitive to the number of training queries than the rest of the methods.

At first glance, the results in Figure 6.25 may appear unsatisfactory. This is explained by the fact that test queries access a substantial portion of the dataset: 24.50% for *SIFT50M*, 41.78% for *SIFT10M*, 36.81% for *SIFT*, and 50.73% for *GIST*. Given a memory budget of 30%, full coverage of all visited nodes is unattainable for all datasets but *SIFT50M*. Nevertheless, the general trend persists: HKPR consistently outperforms other methods, even with very limited training queries, and achieves nearly identical outcomes to those obtained when substantially more training data is used.

In *SIFT* and *SIFT50M*, we observe that EVS and EVSI using 9 (resp. 450) queries outperform the same methods with 45 (resp. 2250) training queries. However, since this anomaly is limited to these 2 datasets (with the difference in *SIFT50M* being very small) and methods that do not have a strong generalization ability, we regard it as statistically insignificant, most likely the result of chance. In this case, the 9 (resp. 450) sampled queries happened to capture priority relationships between nodes more consistent with the test set than the 45 (resp. 2250) query sample. This is yet another example of the dependency of such methods in the actual training set. Small variations in this set might drastically change the results.

In summary, given that both training and test queries are drawn from the same distribution, we conclude that **PPR, and especially HKPR, can deliver strong results with substantially fewer training queries than competing methods**, while also exhibiting far lower sensitivity to the training set size. This characteristic is particularly valuable in practice, enabling systems to achieve robust performance while relying on training sets much smaller than the query loads they are expected to serve, which in theory would also allow them to adapt quicker to shifting workloads.

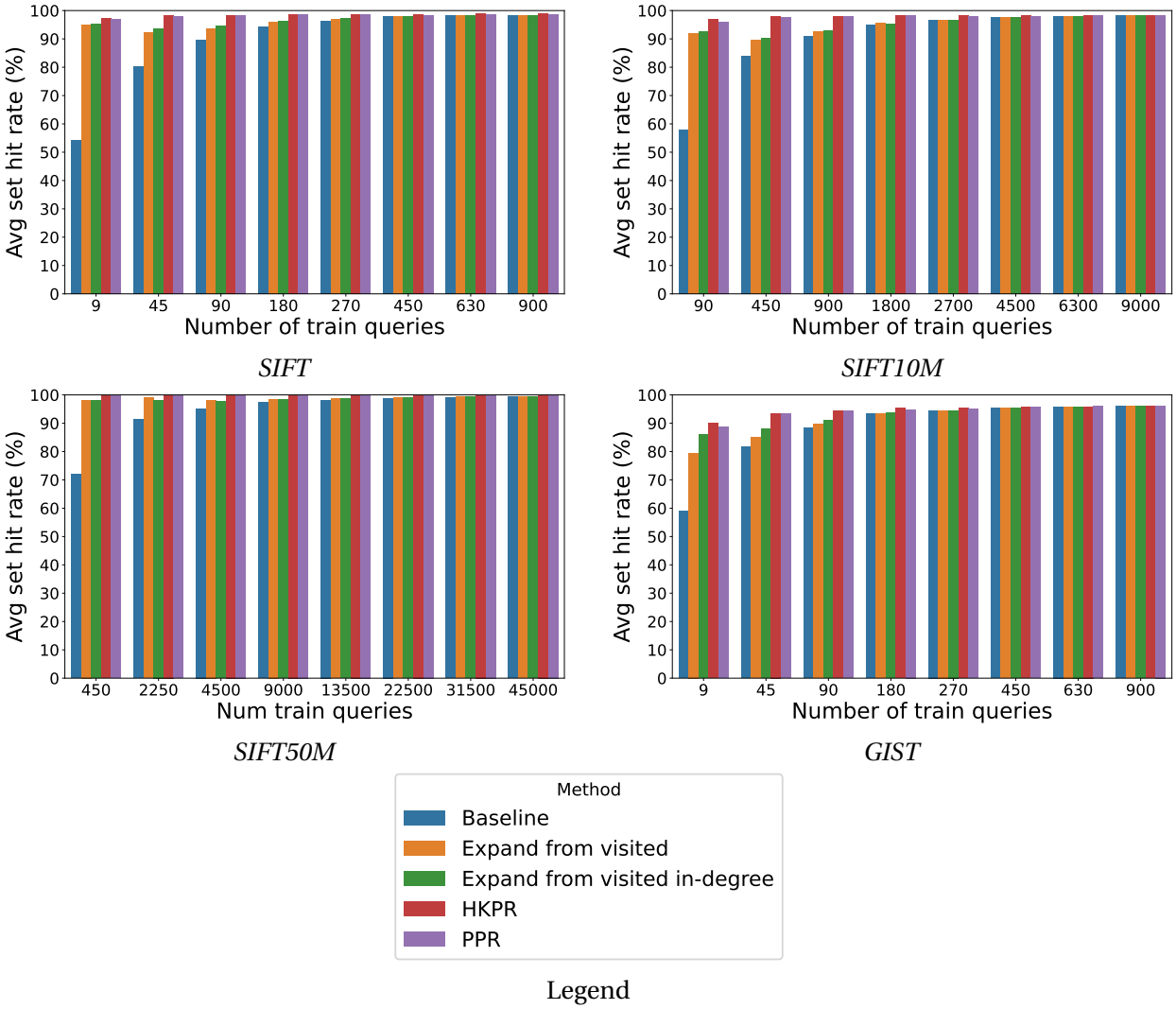


Figure 6.23: Barplot of average set in-memory hit rate over the number of train queries. For workload: $L = 3000$ for *SIFT* and *GIST*, and $L = 30000$ for *SIFT10M* and $C = 1$. *top-K* = 10, PPR $alpha = 0.05$, HKPR t = 2, *memory budget* = 30%.

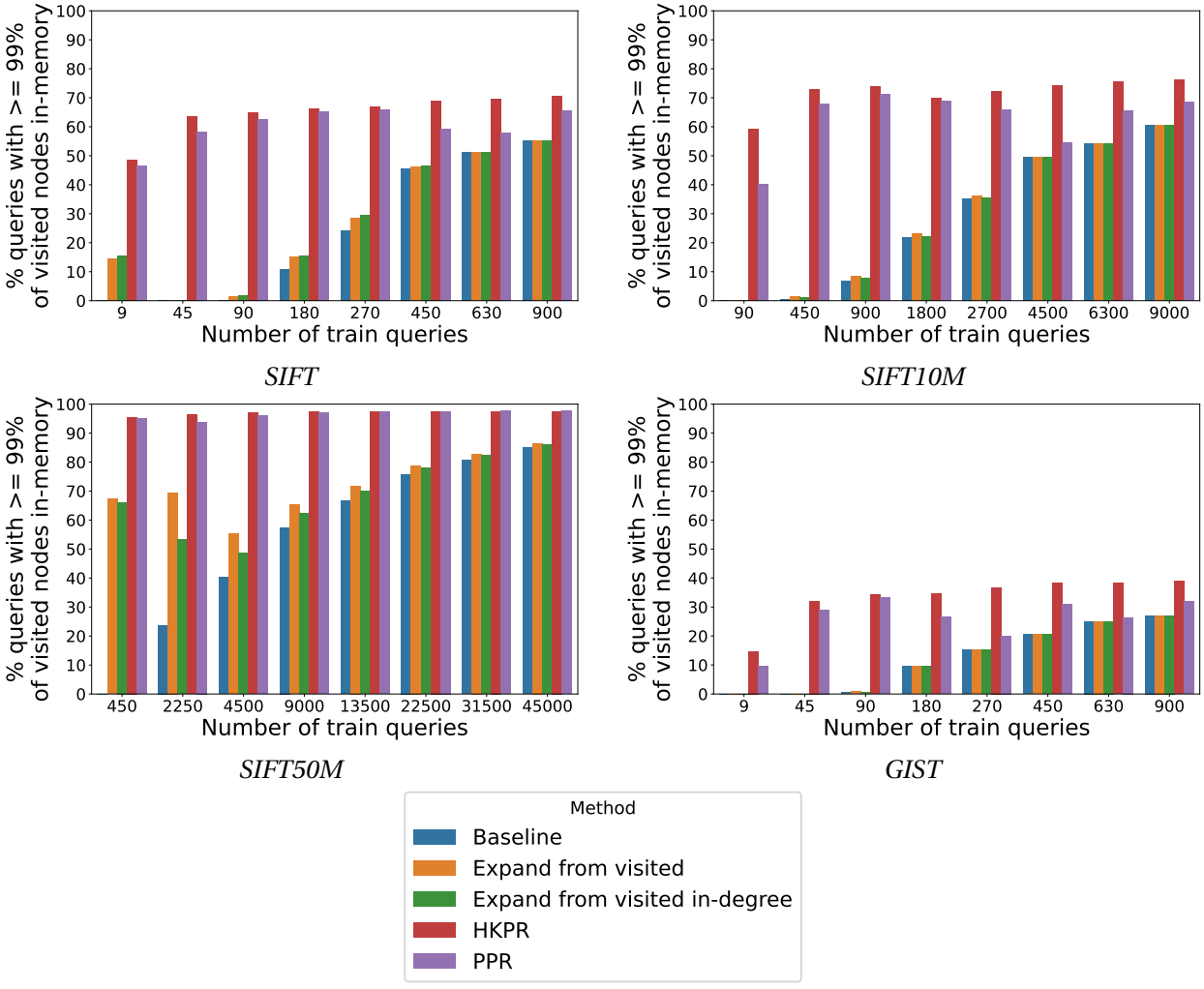


Figure 6.24: Barplot of the percentage of queries with at least 99% of the accessed nodes in memory over the number of train queries. For workload: $L = 3000$ for *SIFT* and *GIST*, and $L = 30000$ for *SIFT10M* and $C = 1$. *top-K* = 10, PPR $alpha = 0.05$, HKPR t = 2, *memory budget* = 30%.

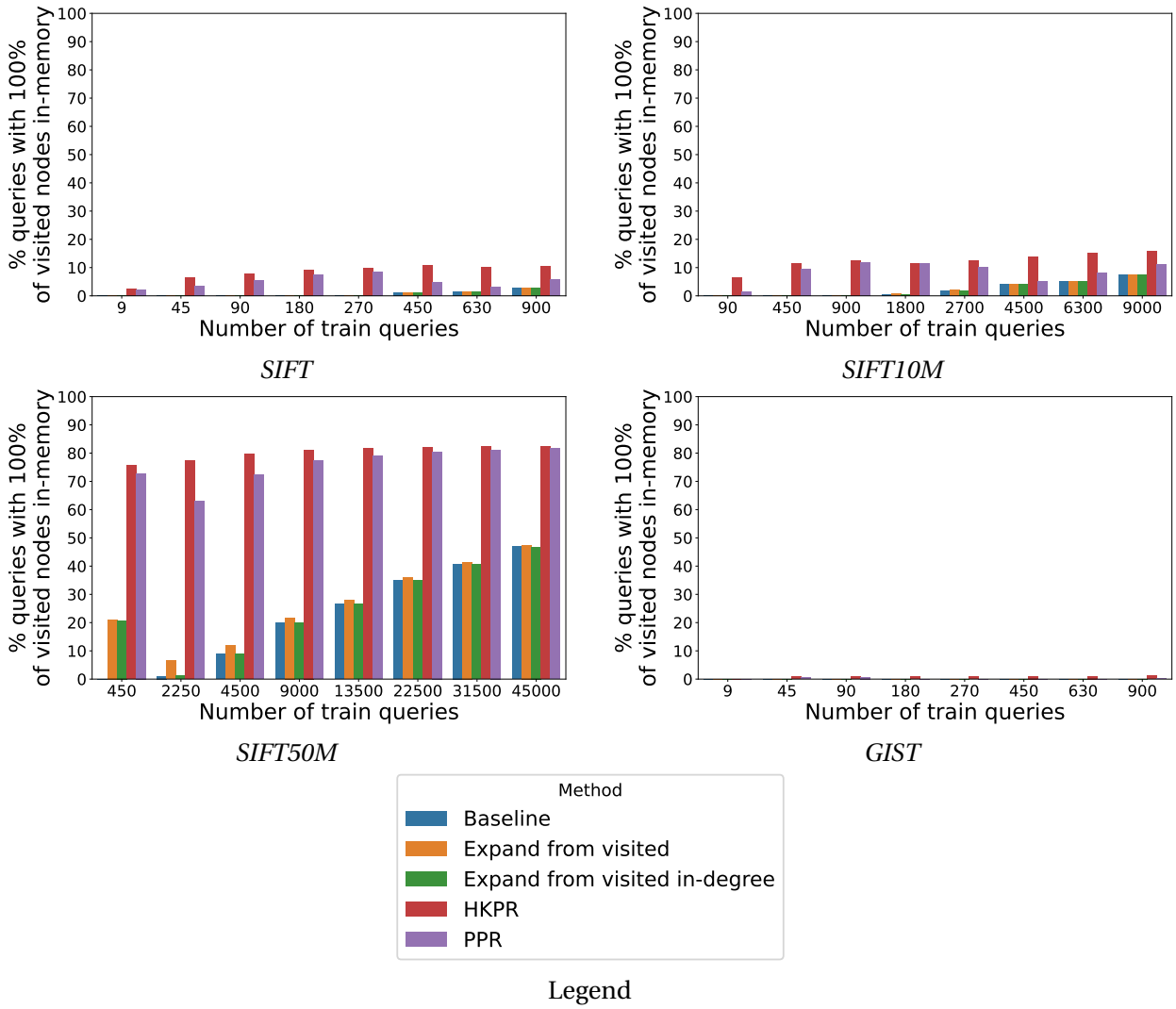


Figure 6.25: Barplot of the percentage of queries with all of the accessed nodes in memory over the number of train queries. For workload: $L = 3000$ for *SIFT* and *GIST*, and $L = 30000$ for *SIFT10M* and $C = 1$. *top-K* = 10, PPR $alpha = 0.05$, HKPR t = 2, *memory budget* = 30%.

6.3.9 Improvement over cluster diameter

In this subsection, we examine how the results vary when one of the workload characteristics — the cluster diameter — changes. We employ different values of L (while keeping $C = 1$ for all experiments) to evaluate the influence of cluster diameter on the performance of different methods. Figure 6.26 illustrates the improvement in the average in-memory set hit rate as L increases, whereas Figure 6.27 shows the percentage of queries for which at least 99% of the accessed vectors are stored in memory as L increases.

For both metrics, two distinct cases can be observed:

1. For *GIST* and *GloVe100*, when $L = 100$, PPR and HKPR achieve substantially higher performance compared to the other approaches. As L increases, the number of available training queries also grows. Consequently, even though more unique vectors are accessed, the training signal becomes both stronger and more general due to the larger set of training elements. Since the other methods rely heavily on the training data and lack strong generalization capabilities, they also benefit from an increase in L — despite the constant *memory budget* and the larger number of accessed vectors. Therefore, as L grows, the performance gap between HKPR and PPR and the other methods narrows.

2. For *SIFT*, *SIFT10M* and *SIFT50M*, when $L = 100$, all methods perform equally well. This outcome is due to the fact that $|UVis_Q^{TEST}|$ is small relative to Θ, and the datasets are less challenging compared to the others. However, increasing L enlarges $|UVis_Q^{TEST}|$, bringing it closer to Θ and thereby reducing the margin for selecting irrelevant nodes. In this context, for larger L, the superior generalization ability of HKPR and PPR allows them to more effectively determine which nodes to retain in memory, thereby widening the performance gap relative to the other methods.

Nonetheless, across all datasets, **increasing L leads to a decline in overall performance** — an expected trend, given that Θ remains constant while $|UVis_Q^{TEST}|$ grows with L.

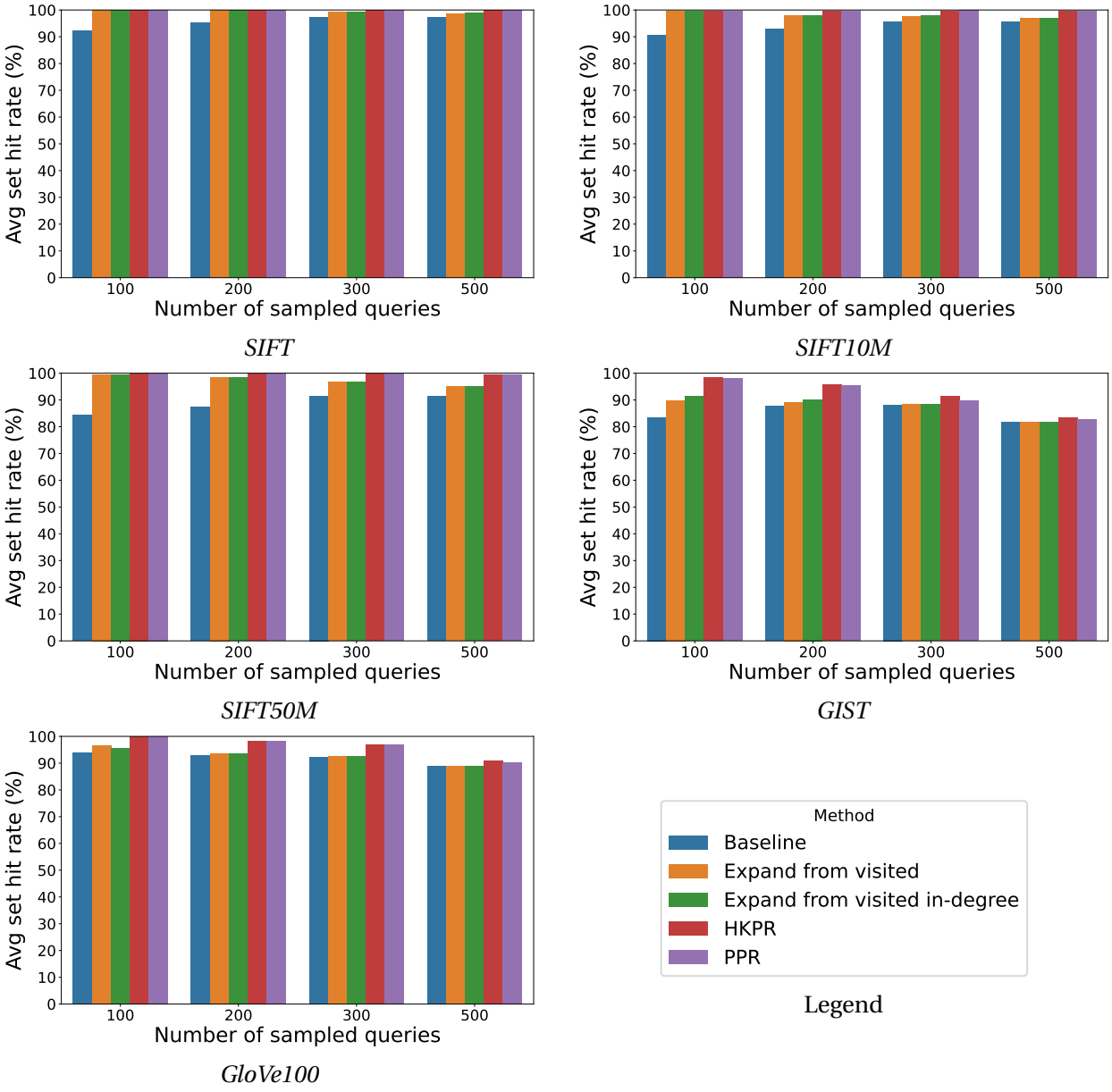


Figure 6.26: Barplot of average set hit rate over L. For workload: $C = 1$. *top-K* = 10, *ef_Search* = 256, PPR $\alpha = 0.05$, HKPR t = 2. For *SIFT10M*, *memory budget* = 10%, for *SIFT50M*, *memory budget* = 5%, while for the rest *memory budget* = 30%

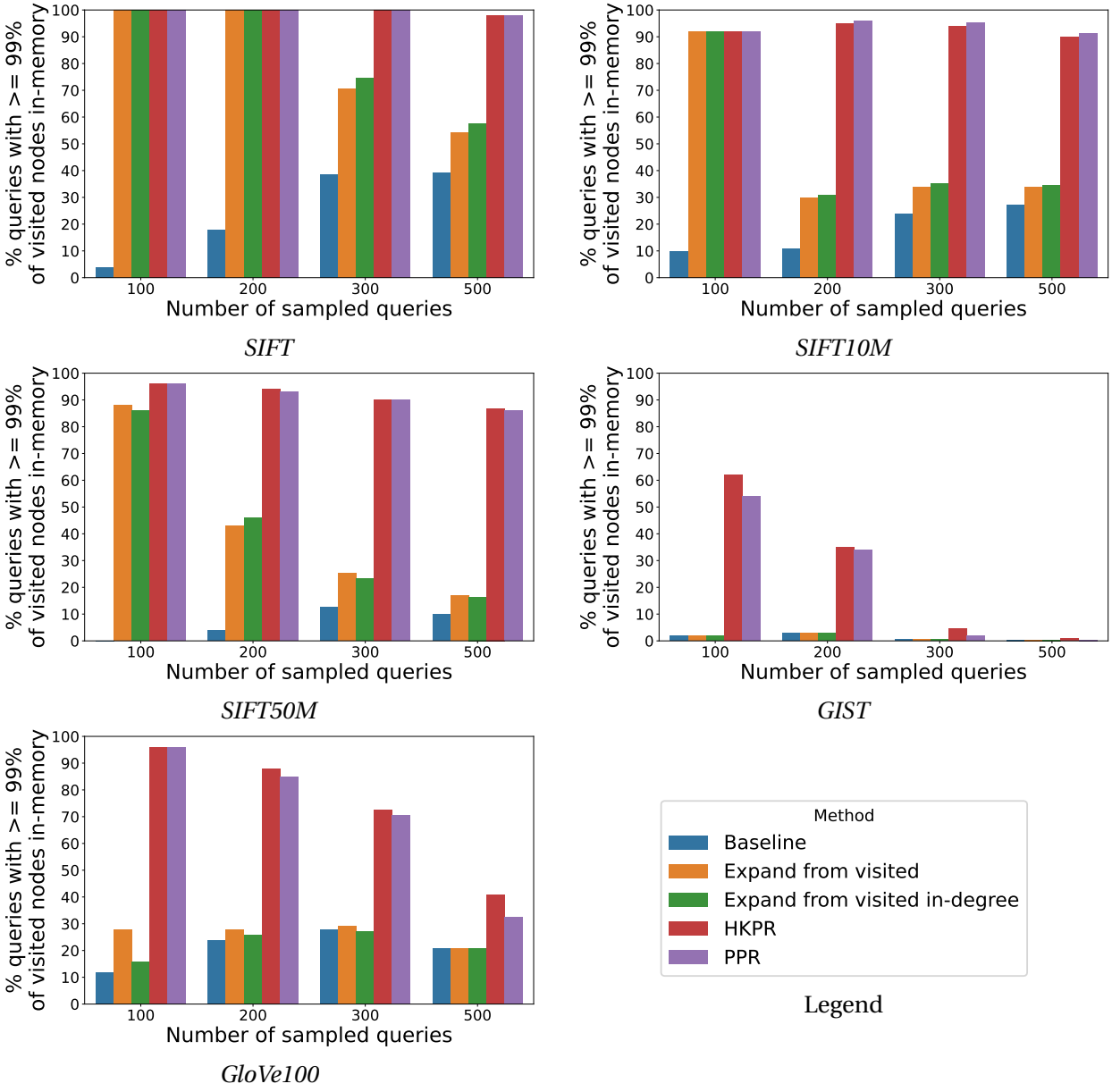


Figure 6.27: Barplot of the percentage of queries with at least 99% of the visited nodes in memory (y-axis) over L (x-axis) . For workload: $C = 1$. *top-K* = 10, *ef_Search* = 256, PPR $alpha = 0.05$, HKPR t = 2. For *SIFT10M*, *memory budget* = 10%, for *SIFT50M*, *memory budget* = 5%, while for the rest *memory budget* = 30%

6.3.10 Improvement over number of clusters

In this subsection, we examine how the results vary when one of the workload characteristics — the number of clusters — changes. We employ different values of C (while keeping $L = 100$ for *GIST* and *SIFT*, and $L = 1000$ for *SIFT10M*, for all experiments) to evaluate the influence of

the number of clusters on the performance of different methods. For this experiment, we used the learn split, as it provides more queries and thus allows to test a bigger number of clusters. Figure 6.28 illustrates the improvement in the average in-memory set hit rate as C (the number of clusters) increases, whereas Figure 6.29 (resp. Figure 6.30) shows the percentage of queries for which at least 99% (resp. 100%) of the accessed vectors are stored in memory as C increases.

As the number of clusters increases and L stays the same, the total amount of workload queries increases, and so does the percentage of the dataset visited. These queries visit 10.53%, 7.87%, 18.59% of the dataset with $C = 1$ for *SIFT*, *SIFT10M*, *SIFT50M*, but visit 33.41%, 36.68%, 34.35% of the dataset with $C = 6$, respectively.

As the *memory budget* stays the same, the average set in-memory rate as well as the percentage of queries with 99% and 100% of the accessed vectors in memory decrease, since there are much more accessed nodes with $C = 6$ compared to $C = 1$. Nevertheless, in Figure 6.29 and Figure 6.30, there are some exceptions to this rule, specially for $C = 6$. This might have happened because spheres of visited nodes around 2 clusters of queries might have merged, and some nodes are visited by queries belonging to 2 clusters.

Once again, PPR and HKPR show superior results, with smaller degradation of results as C increases compared to the other methods. The reason is once again the same: superior ability of generalization of results seen during training. These results would be even more superior compared to the other methods if a smaller training split, opposed to the 50/50 used in this experiment, which benefits the baseline, EVS and EVSI. As shown in subsection 6.3.8, HKPR and PPR show a much smaller degradation of results as the number of train queries decrease compared to the number of testing queries.

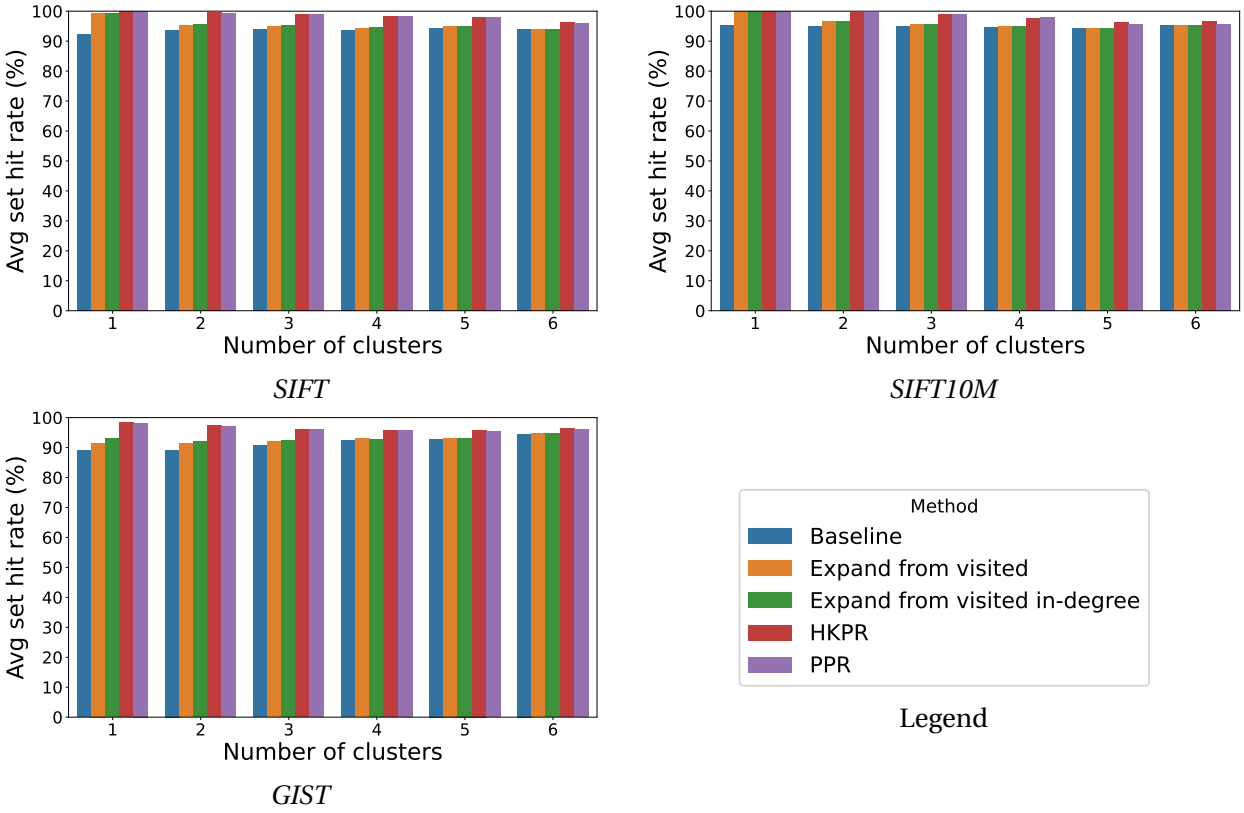


Figure 6.28: Barplot of average in-memory set hit rate over C. For workload: $L = 100$ for *GIST* and *SIFT*, and $L = 1000$ for *SIFT10M*. *top-K* = 10, *ef_Search* = 256, PPR $\alpha = 0.05$, HKPR t = 2, *memory budget* = 30%

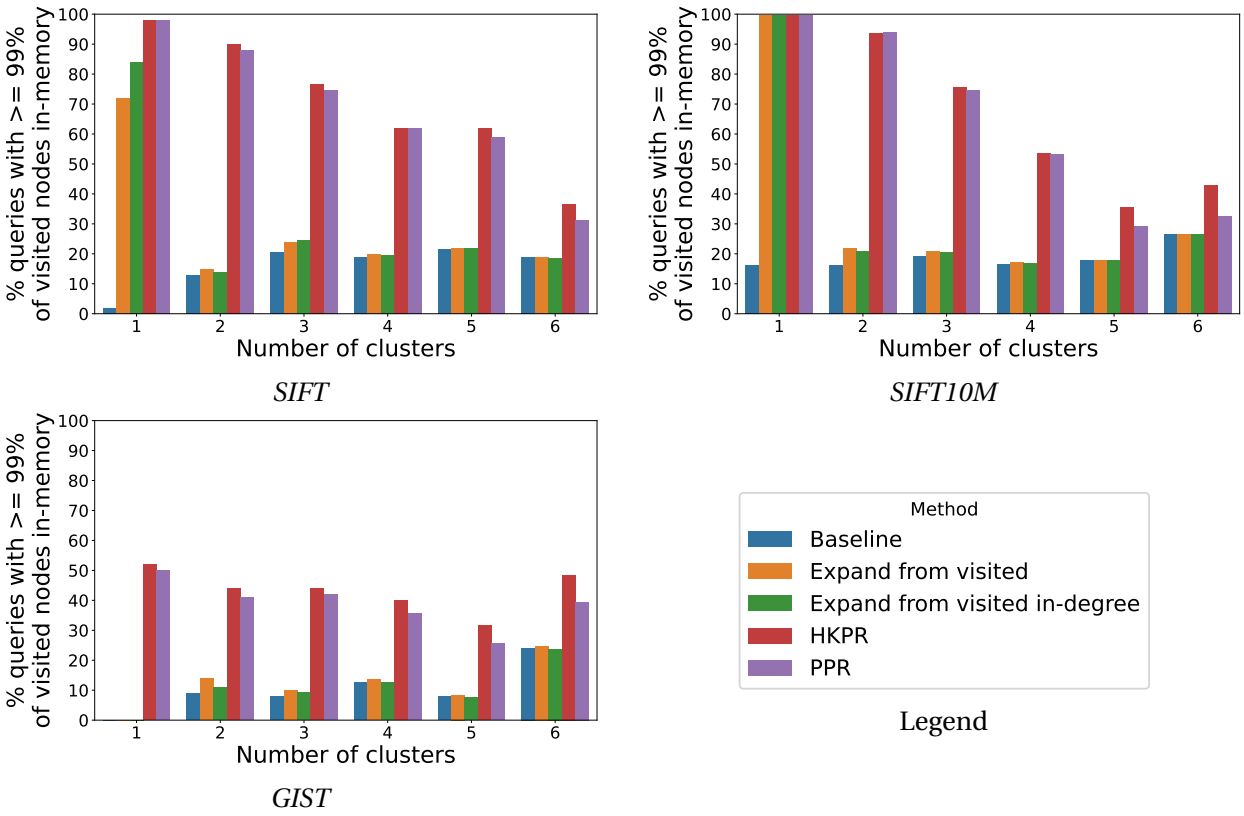


Figure 6.29: Barplot of percentage of test queries with at least 99% of accessed nodes in memory over C. For workload: $L = 100$ for *GIST* and *SIFT*, and $L = 1000$ for *SIFT10M*. *top-K* = 10, *ef_Search* = 256, PPR $\alpha = 0.05$, HKPR t = 2, *memory budget* = 30%

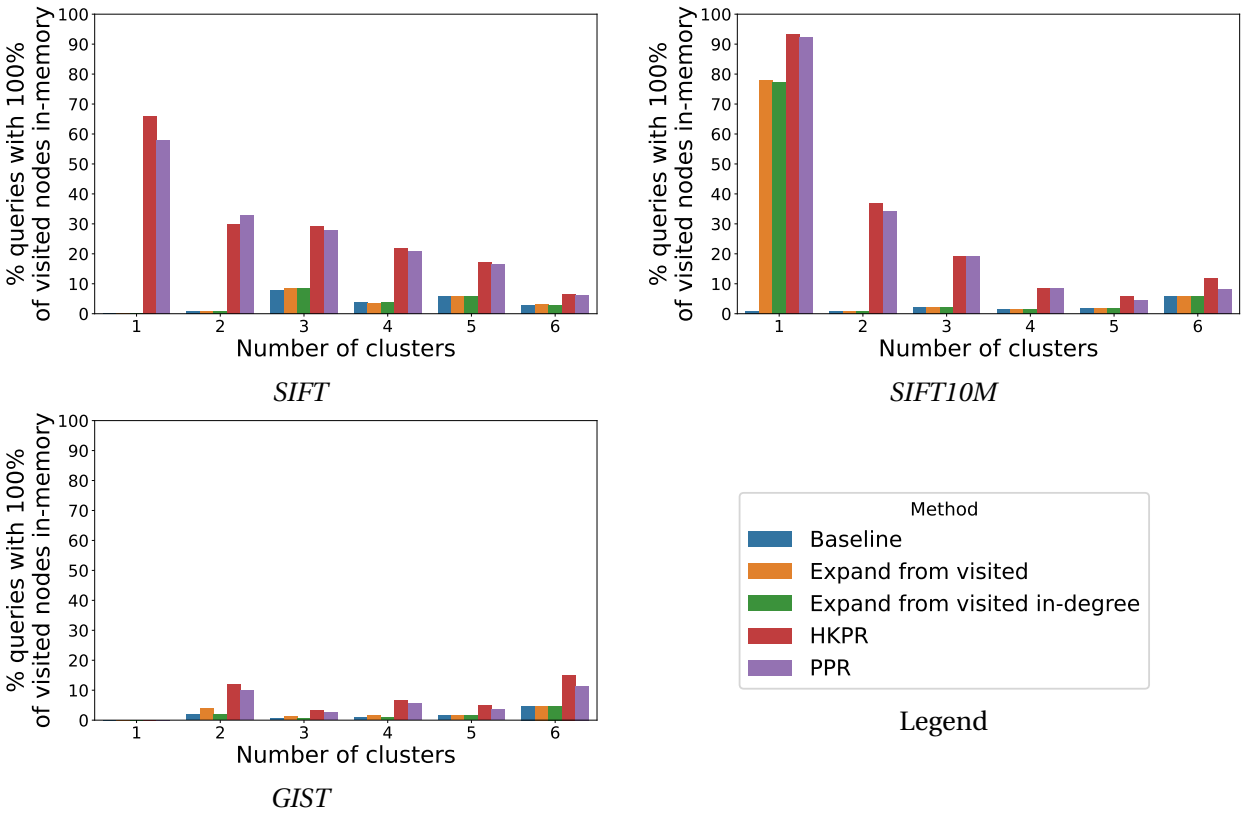


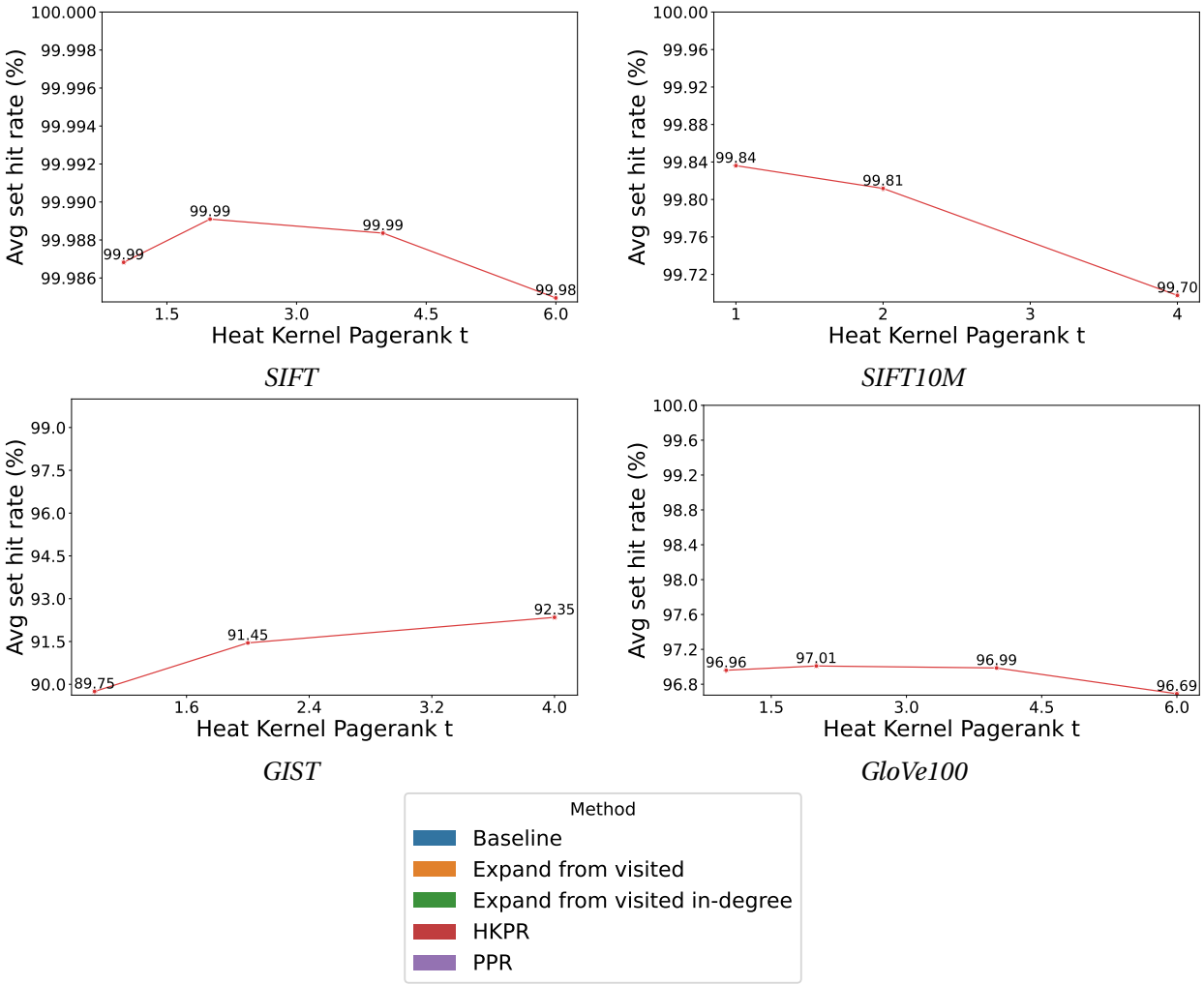
Figure 6.30: Barplot of percentage of test queries with all accessed nodes in memory over C. For workload: $L = 100$ for *GIST* and *SIFT*, and $L = 1000$ for *SIFT10M*. *top-K* = 10, *ef_Search* = 256, PPR $\alpha = 0.05$, HKPR $t = 2$, *memory budget* = 30%

6.3.11 Finetuning HKPR and PPR

In this section, we provide a concise explanation regarding the chosen value for the HKPR parameter t and PPR parameter α. Various combinations of L and C were evaluated, along with both metrics introduced above. However, for brevity, we limit our presentation to the results obtained with $L = 300$ and $C = 1$. We do not claim that this selection is optimal across all datasets and scenarios. Rather, we demonstrate that adopting $t = 2$ and $\alpha = 0.05$ constitutes a valid choice in certain contexts and sufficiently illustrates the promising potential of HKPR and PPR for the task under consideration.

As depicted in Figure 6.31, it is evident that increasing t generally leads to deteriorated results. This observation aligns with expectations: with larger t, the initial values diffuse more widely to nodes situated farther from any node in $UVis_{\mathcal{Q}}^{TRAIN}$. Given the assumed workload and the premise that training queries are representative of this workload, such diffusion is undesirable for our application. Although there may be an initial performance improvement — motivating the choice of $t = 2$ instead of $t = 1$ — this is followed by a subsequent decline in performance.

For the sake of brevity, we omit a similar subsection for PPR. Nonetheless, the underlying principle remains consistent: as the parameter α increases, the probability that a random walker continues its path — rather than returning to a node in $UVis_{\mathcal{Q}}^{TRAIN}$ — also increases, causing the score to spread more broadly to distant nodes. Again, this effect is unfavorable for our application. While the typical value of α in PPR applications is approximately 0.85 [3, 31, 49], in our context, the acceptable range for α is considerably lower, as performance rapidly declines with increasing α. Consequently, we opted for $\alpha = 0.05$, although alternative values may also be suitable.



Legend

Figure 6.31: Lineplot of average set hit rate over HKPR t parameter. For workload: $L = 300$ and $C = 1$. *top-K* = 10, *ef_Search* = 256. For *SIFT10M*, *memory budget* = 10%, for *SIFT50M*, *memory budget* = 5%, while for the rest *memory budget* = 30%

6.3.12 Comparison with DiskANN

Lastly, in this subsection, we present a comparison with another hybrid indexing system, DiskANN [24]. Since we do not account for different in-memory vector storage layouts, the fairest basis for comparison with DiskANN is the number of vectors that must be fetched from disk for each query. Depending on the storage layout, there exists the possibility of *piggybacking* additional vectors located close to the target vector on disk. Because the latency of reading a single vector from memory is approximately the same as reading multiple vectors stored contiguously, *piggybacking* attempts to exploit this fact to pre-fetch potentially relevant vectors, thus reducing the total I/O cost. As our methods could also benefit from such techniques, we configure the comparison to be as fair as possible by setting the disk beam width to 1. This ensures that DiskANN performs exactly one disk I/O operation per vector.

It should be noted that our approach begins at a relative disadvantage, since DiskANN requires fewer total vectors resident in memory. According to the DiskANN paper, the search procedure is guided by distance computations over quantized vectors that are held entirely in memory. When DiskANN needs to fetch a node's neighbor list from disk, the corresponding full-precision vector — stored contiguously with the list — is retrieved in the same I/O operation. This full-precision vector is then only used in the final re-ranking stage, which occurs after all necessary full-precision vectors have been retrieved. The cost of keeping all the quantized vectors in memory can itself be substantial. In our experiments, quantized vectors for *SIFT10M* occupy around 1.1 *GB*. Were the same number of bytes per vector be used with *SIFT50M*, this cost would amount to 5.5 *GB*. Therefore, our caching techniques could also be used to cache part of the quantized vectors in memory, while keeping the rest on disk.

In other words, **DiskANN only fetches from disk those vectors belonging to the search path, not all visited vectors**. Recall that *nodes on the search path* are those inserted in the priority queue during search (see section 2.4), whereas *visited nodes* comprise all nodes for which distance computations were performed, including neighbors that were considered but not inserted into the queue. Therefore, the minimal in-memory set required to answer all queries without disk I/O is considerably larger for our approach than for DiskANN.

For our evaluation, we extended the `diskannpy` wrapper [57] to expose the statistics collected internally by the DiskANN C++ engine (via the `diskann::QueryStats` class in `static_disk_index.cpp`), which are not available in its Python interface. Both the wrapper and the C++ engine were built from source [57], at commit `1f08328e25f2476286352bb93de19c158aef1fea`. The index was built on the base set using `build_disk_index`, with `complexity` and `graph_degree` parameters (analogous to *ef_Construction* and M in HNSW) set to 300 and 64, respectively (corresponding to $2M$ in layer 0 of the HNSW index).

We then executed an identical workload: first the training queries, followed by the test queries from which statistics were collected. The *memory budget* parameter directly determines the

value of `num_nodes_to_cache` when instantiating `StaticDiskIndex` in `diskannpy`. In DiskANN, the search parameter `complexity` corresponds to *ef_Search*. For this experiment, we used the *SIFT*, *SIFT10M*, and *GIST* datasets; *GloVe100* was excluded because DiskANN does not support cosine distance for static disk indexes.

Figure 6.32 and Figure 6.33 depict the evolution of the average number of vectors accessed on disk per query as a function of *memory budget* and *ef_Search*, respectively. DiskANN supports two caching mechanisms:

- The default cache (`cache_mechanism` = 1), initialized with nodes sampled when the index was created. This corresponds **to the brown bars in the plots**.
- An alternative cache mode (`cache_mechanism` = 2), which reserves capacity for up to `num_nodes_to_cache` entries without initializing them. This corresponds **to the pink bars in the plots**.

Our first observation is that, beyond a certain *memory budget* threshold, **HKPR consistently outperforms both of DiskANN's caching mechanisms — even though our methods must manage a significantly larger total number of vectors.** Because we require more vectors in memory, our methods necessarily access more disk vectors at lower *memory budget* values compared to DiskANN.

Nonetheless, unlike our methods, DiskANN is oblivious to workload-specific access patterns and instead caches a representative sample of the entire dataset. This uniform sampling strategy works well for broad coverage but is suboptimal for workloads concentrated in specific regions of the search space. In contrast, as *memory budget* increases, our caching policies adaptively retain vectors clustered in the "hot" regions of the search space, while DiskANN disperses cache allocations more uniformly.

Moreover, as dataset size increases (e.g., comparing *SIFT* to *SIFT10M*), our methods produce an even larger performance gain relative to DiskANN. The reason lies in DiskANN's uniform sampling: as the dataset grows, the probability that a cached vector is actually used decreases. On the other hand, caching the same percentage of *SIFT10M* and *SIFT* allows for far more vectors to be cached in the former case than in the latter. Since the fraction of vectors actually accessed does not grow linearly with dataset size (see subsection 6.3.4), our workload-aware caching scales more effectively. Thus, for larger datasets, the *memory budget* required to surpass DiskANN becomes proportionally smaller. We expect this trend to persist for even larger datasets.

When examining performance as a function of *ef_Search*, we observe that for both *SIFT* and *SIFT10M*, the number of vectors accessed from memory grows much faster in DiskANN than in our methods (excluding the baseline). Raising *ef_Search* increases the total number of accessed vectors, but in our case, it also strengthens the learning signal for the caching policy, resulting in more intelligent caching and a slower growth rate in disk accesses. On the other hand, the

workload-agnostic approach of DiskANN means that it does not take advantage of this increased learning signal and thus is more affected by the increase in total number of accessed vectors. This effect is particularly pronounced for *SIFT10M*.

The *GIST* dataset follows a somewhat different trend. While our methods eventually surpass DiskANN at sufficiently high *memory budget* (around 60%), DiskANN's uniform sampling is more effective here due to the dataset's more evenly distributed query access pattern. With only 1000 queries, selecting 300 for training covers a substantial and relatively uniform portion of the dataset (as hinted by the heatmaps in chapter 4). This observation, coupled with the fact that DiskANN needs less full-precision vectors in memory overall, likely explains our reduced advantage for most *memory budget* values in this particular case.

However, if we apply the approach described in subsection 6.3.7 — either ignoring on-disk nodes for all queries or via a per-query heuristic — the total number of accessed vectors drops substantially, potentially allowing us to outperform DiskANN even on *GIST*. Importantly, these results must be interpreted in light of the fact that **DiskANN sacrifices recall by performing graph search solely using quantized vectors in memory, resorting to full-precision vectors only during final re-ranking**. Consequently, ignoring on-disk nodes in our method may still yield recall equal to or greater than DiskANN. A detailed study of recall differences in such configurations is left for future work.

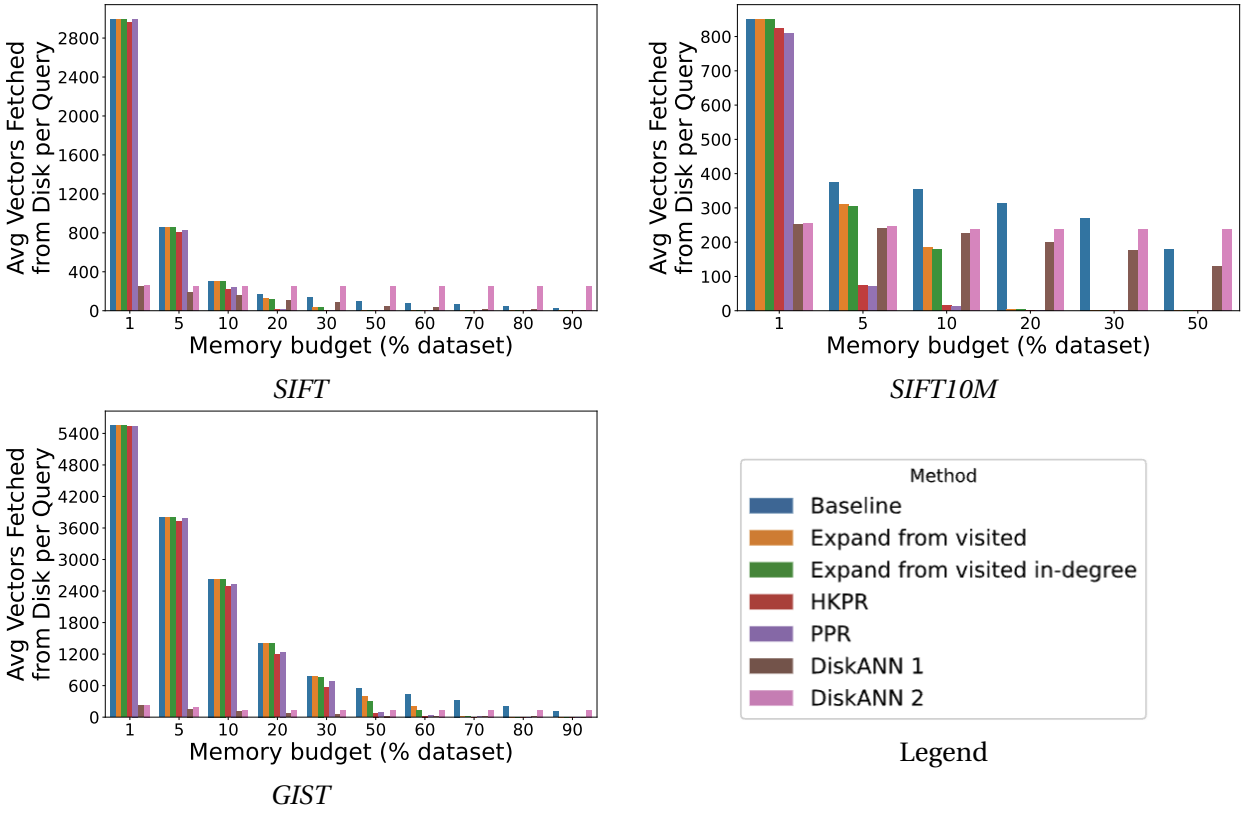


Figure 6.32: Barplot of average number of accessed vectors on disk, per query, over *memory budget*. For workload: $L = 300$ and $C = 1$. *top-K* = 10, *ef_Search* = 256, PPR $\alpha = 0.05$, HKPR t = 2.

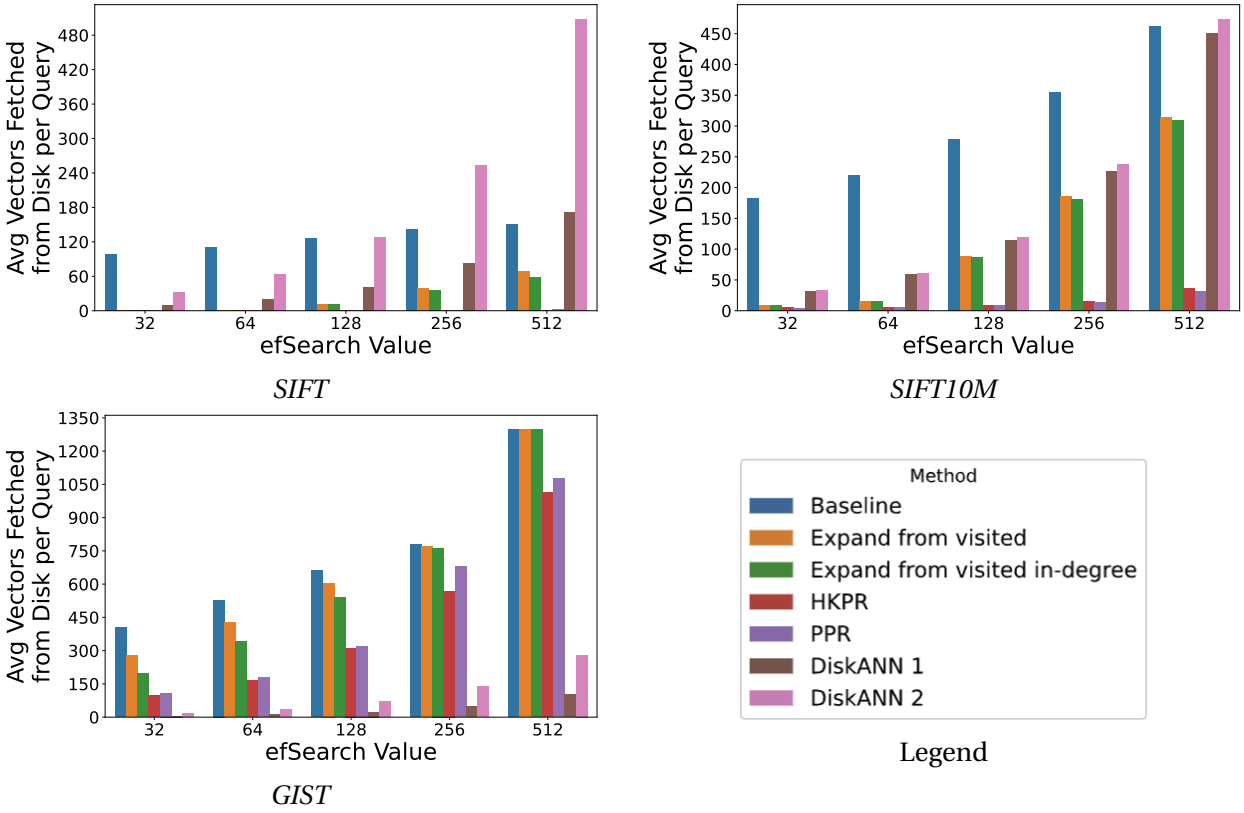


Figure 6.33: Barplot of average number of accessed vectors on disk, per query, over *ef_Search*. For workload: $L = 300$ and $C = 1$. *top-K* = 10, PPR $alpha = 0.05$, HKPR t = 2. For *SIFT10M*, *memory budget* = 10%, while for the rest *memory budget* = 30%

6.4 Conclusion

In this chapter, we introduced a novel approach to reducing the memory footprint of the HNSW index, explicitly leveraging workload knowledge and tailoring it to clustered query workloads. Our method relies on a caching mechanism in which a subset of training queries is used to assign cache priorities to all nodes in the graph. Based on the specified *memory budget*, the vectors with the highest cache priority are retained in memory, while the remaining vectors are stored on disk.

We proposed and evaluated several cache priority policies: Most Frequently Used (MFU) (baseline), Expand from the Visited Set (EVS), Expand from the Visited Set In-Degree (EVSI), Personalized PageRank (PPR), and Heat Kernel PageRank (HKPR). Each policy was described in detail, including the underlying intuition, advantages, and limitations. We then conducted an extensive evaluation, comparing these policies across multiple metrics, experimental setups, and workloads.

Our results demonstrate that even when the in-memory set hit rates of the cache policies are comparable, both PPR and, in particular, HKPR are able to serve significantly more queries without incurring I/O operations. These two policies consistently outperform the baseline, EVS and EVSI across diverse datasets, *ef_Search* values, and *memory budget* configurations.

Since most testing queries are unseen during training, they traverse a node set that differs from the ones observed in the training phase. Consequently, cache policies such as the baseline and EVS, which overly rely on training results without sufficient generalization, fail to accurately predict node visits for unseen queries, even from the same workload. This limitation becomes increasingly problematic as the volume of test queries exceeds that of the training set. By contrast, PPR and HKPR demonstrate much greater robustness to limited training data, maintaining performance with only minimal degradation as the number of training queries decreases. Notably, HKPR achieves results with a small training set that are close to those obtained with a substantially larger training set, and better than those obtained with any number of training queries by the baseline, EVS, EVSI, for some configurations of *memory budget* and *ef_Search*.

We further observed that both the average in-memory set hit rate and the proportion of queries with at least 99% of their accessed nodes residing in memory increase with larger *memory budget* values. However, HKPR consistently delivers superior results at smaller memory budgets compared to the baseline and EVS, up to the point of saturation where most policies converge in performance. For a fixed *memory budget*, the impact of *ef_Search* varies: performance may improve when additional nodes overlap with training data, or decline when excessive exploration reduces in-memory coverage.

Additionally, we assessed the impact on recall of ignoring vectors stored on disk and restricting the search to the in-memory subgraph. We showed that, for queries with at least 95% of their accessed vectors in memory, the loss in recall is negligible. Therefore, these disk-resident vectors can be safely ignored when prioritizing latency, depending on the desired recall-latency tradeoff. Since HKPR's priorities allow for far more queries to have most of the vectors in memory, it is able to answer much more queries without any I/Os, conserving the latency of an in-memory HNSW index.

Moreover, we investigated the impact of increasing the number of clusters in the results. Generally, results worsen as the number of cluster increases, since the *memory budget* was fixed and the number of total sampled queries ($C \times L$ assuming no query is chosen twice) also increases, as well as the percentage of the dataset visited by them. Nevertheless, HKPR showed superior performance in all considered metrics, having its resuls less deteriorated as the number of clusters increases, outperforming all other methods.

We also investigated the effect of cluster diameter on method performance. As the diameter increases, two scenarios arise: if results are already saturated at a smaller diameter, widening it reduces the performance of the baseline, EVS, and EVSI relative to PPR and HKPR. Otherwise, the performance gap tends to decrease. Nonetheless, PPR and especially HKPR consistently outperform the other methods, particularly in serving a higher number of queries with minimal

or no I/O operations.

Returning to the research questions posed in section 3.2:

MEMORY_Q1 To determine which nodes to keep in memory, use a representative set of training queries that reflects the expected workload. Record the frequency with which each node is visited, and input this information into HKPR, which then assigns a cache priority to each node. According to the available *memory budget*, retain the vectors with the highest priorities.

MEMORY_Q2 Increasing *memory budget* allows more queries to be answered completely from memory. The rate of improvement, however, varies across methods. HKPR exhibits substantial performance gains at smaller memory budgets, achieving superior results earlier than competing methods. Although a larger *ef_Search* makes search to visit more nodes and can reduce overall cache efficiency for a fixed *memory budget*, it may slightly improve the performance of weaker methods - such as the baseline, EVS and EVSI - by increasing the overlap between vectors accessed by train and test queries.

MEMORY_Q3 While the proportion of queries resolved entirely in memory depends on the method, dataset, *ef_Search*, *memory budget*, and workload properties, we demonstrated that with only a 5% memory budget, HKPR successfully answers up to 50% of the queries in the *SIFT50M* dataset, whereas the baseline and EVS fail to answer any under the same configuration. Furthermore, if a small loss of recall is acceptable in exchange for significant latency benefits — a compromise commonly adopted in hybrid indexing approaches — nodes on disk can be ignored for queries with at least 95% of their nodes cached in memory (and potentially for even smaller fractions, depending on the dataset and workload). Such queries can be identified through heuristics, which we leave as future work. In this scenario, HKPR is able to answer at least 72% of the queries entirely from memory under a 5% *memory budget* on *SIFT50M*, while other methods remain below 30%.

Chapter 7

Related Work

In this chapter, we review existing methods for addressing the issues mentioned in chapter 3, analyzing how they differ from our proposed solution.

7.1 Latency: Improving Entry Point Selection

The notion that HNSW yields only marginal benefit over NSW for high-dimensional vectors and large *ef_Search*- as discussed in this work - is not new. For instance, one study [44] finds that, for high-dimensional embeddings, a flat NSW graph matches HNSW in both latency and recall. The authors further observe that in high dimensions, a small set of *hub* nodes - which are visited disproportionately often and possess high in-degrees - naturally form a well-connected "highway" for searches, reducing the necessity for skip-list based hierarchy. Moreover, using a single flat graph confers additional memory savings relative to HNSW. Nonetheless, because these hubs are more frequently visited than other nodes, they are also good candidates for upper-layer selection in HNSW.

Other studies [14, 45] report that HNSW often fails to deliver logarithmic search complexity in high-dimensional datasets, primarily due to exhaustive search within each layer. These works also reveal a link between node insertion order and the *Local Intrinsic Dimensionality* (LID) of each vector, and argue that preferentially inserting nodes with higher LID into the upper layers can boost recall by as much as 12.8 percentage points. However, such methods are workload-agnostic: they do not exploit any knowledge of workload distribution, implicitly assuming all regions of vector space are equally likely to be queried.

7.2 Memory: Reducing the Memory Footprint

A key challenge in large-scale vector search lies in mitigating memory consumption. Two main directions have emerged in recent research: (i) lossy vector transformations to reduce dimensionality, and (ii) disk-oriented indexing structures. The first approach relies on quantization techniques [72], while the second leverages disk-based indexes, which may be graph-based [24, 54, 58] or follow alternative designs [6, 71].

Disk-Oriented Graph Indexes. One of the most influential disk-based graph structures is *DiskANN*, which introduces the *Vamana* index, based on the RNG concept. Unlike HNSW, Vamana is a single-layered graph, augmented with a parameter α that controls the graph's diameter. While HNSW corresponds to $\alpha = 1$, choosing a larger value ensures multiplicative reductions in query distance at each step, thereby decreasing diameter and improving search efficiency.

DiskANN also proposes a scalable graph construction mechanism for scenarios in which the full dataset cannot reside in memory. The idea is to partition data into k clusters via KMeans [27], assign each point to its l nearest clusters to ensure overlap, and build a Vamana index independently for each cluster. These partial graphs are then merged by uniting their edges. Such a procedure could also be adapted for HNSW, extending its applicability to memory-constrained environments. ScaDANN [18] advances this line of work by introducing block-level insertion and merging strategies that preserve connectivity in memory-restricted graph indexes.

Another recent method [74] replaces DiskANN's KMeans partitioning with a *Boundary-Adaptive Balanced Partition*, leading to improved results. The authors further design a graph that reduces the number of search hops, thereby decreasing I/O operations. Their framework, like DiskANN, keeps quantized vectors in memory while storing full-precision vectors on disk. In contrast to DiskANN, however, they retain a fully in-memory subgraph (roughly 1% of the dataset) to facilitate entry-point selection. Search proceeds greedily on that upper layer, while the disk-resident bottom layer is searched using an in-memory priority queue of candidates, always expanding nodes already in memory to mask disk I/O delays. They call this method a In-Memory First search. This approach is orthogonal to ours, and could in fact be combined with our caching methods for further efficiency gains.

Search and Storage Optimizations. DiskANN employs Product Quantization (PQ) to guide traversal, storing full-precision vectors alongside their neighbor lists on disk to avoid additional reads during re-ranking. At each hop, quantized vectors determine the next hop, while the full vector is fetched from disk alongside the neighborhood list. The full-precision vector is then used at the end of search for result re-ranking. Parallel pre-fetching is governed by a tunable disk beam width, with higher values reducing I/O requests by anticipating future node fetches. Furthermore, DiskANN supports caching strategies, typically storing nodes $C = 3$ or 4 hops away

from the entry point. Nevertheless, this fixed policy ignores workload characteristics, preventing it from answering many queries without disk I/O under constrained memory. By contrast, our approach exploits workload knowledge, making it possible to avoid I/O more effectively, while guaranteeing recall equivalence to in-memory HNSW if no vectors are ignored during search. Thus, DiskANN's caching policy is suboptimal for our use case, and was shown that DiskANN needs to fetch more vectors from disk than our caching methods, for clustered query workloads (subsection 6.3.12).

Additional efforts extend DiskANN in various ways. FreshDiskANN [58] enables dynamic operations such as concurrent inserts and deletes but leaves the aforementioned DiskANN's limitations unaddressed. DiskANN++ [46] improves entry-point selection by clustering vectors and using the closest medoid to start search. They additionally present a novel storage layout, storing vectors that are close to each other in the same disk page, while also utilizing idle I/O time to expand extra nodes that were fetched in the same block as the requested node. This, however, assumes that more than 1 block can hold more than 1 vector, which might not always be true. Finally, OOD-DiskANN [22] adapts graph construction and quantization techniques to improve robustness for out-of-distribution (OOD) queries, introducing the parallel *ParallelGorder* algorithm for graph reordering. This method could also complement our caching techniques for further I/O reduction.

Alternative Disk-Based Structures. Other approaches adopt different storage paradigms. LSM-VEC [76] builds a disk-based index on top of log-structured merge (LSM) storage, optimizing for high write throughput. Its layered structure places upper layers in memory and the base layer on disk. They propose a sampling-based search method to decide which neighbors to use for next hop selection, trading recall for reduced I/O. By co-locating vectors connected through frequently traversed edges, LSM-VEC enhances disk locality and reduces random I/O operations during graph traversal. Similarly, SPFresh [71] and SPANN [6] are partition-based disk indexes that exhaustively search clusters near the query, but unlike DiskANN, they are not graph-based.

Relation to Our Work. While all of the aforementioned methods aim to minimize I/O operations, few explicitly propose caching techniques. Our approach is complementary: by strategically caching vectors from disk-resident layers, we reduce I/O even further and can be integrated with any of these disk-based frameworks.

Heat Kernel and Personalized PageRank. Aside from indexing methods, related work has addressed efficient estimation of PPR and HKPR. Several algorithms approximate these scores [9, 10, 33, 34, 73]. FAST-PPR applies bidirectional search for pairwise estimation, but is limited to single-source and single-target queries. A more fitting approach [33] introduces a bidirectional estimator with high-probability error guarantees, accelerating computation by 3–8$\times$ over earlier methods and directly obtaining the *top-K* result nodes from arbitrary starting distributions

without iterating over all candidates.

For Heat Kernel PageRank, fewer results are available. To our knowledge, no method focuses exclusively on computing the *top-K* nodes with highest HKPR scores from a general probability distribution. The *ApproxHKPRseed* algorithm [73] extends single-source HKPR computations to arbitrary distributions using random walk sampling. Additionally, Dirichlet Heat Kernel PageRank [10] enables random walks on induced subgraphs, which is also applicable in our setting.

In summary, while prior work has extensively reduced memory overhead via quantization, disk-based graph indexing, and storage optimizations, explicit caching strategies remain underexplored. This opens opportunities for our approach, which integrates naturally with these methods to minimize I/O costs without compromising recall.

Chapter 8

Conclusion

In this thesis, we explore techniques for scaling the HNSW index by leveraging workload information. Although HNSW is considered *state-of-the-art*, it exhibits limitations as dataset sizes grow. Modern information retrieval systems often need to store millions or even billions of vectors. For datasets of such magnitude, retaining all vectors and their associated graph metadata in main memory quickly becomes infeasible.

In real industrial workloads, query access patterns are typically skewed and clustered. This implies that certain vectors are accessed significantly more frequently than others, resulting in clusters of activity [36, 42, 43]. Consequently, the majority of vectors are rarely, if ever, accessed. Assuming clustered query workloads, we propose methods to improve both the latency and memory requirements of HNSW.

To evaluate these proposals, we required clustered query workloads. However, standard ANN benchmark datasets do not provide such workloads. Instead, they consist of query sets uniformly distributed across the search space. While these uniform query sets serve as stress tests for indexes, they fail to capture the workload we target. To address this gap, we propose a method to generate clustered query workloads from existing benchmark query sets. We demonstrate the effectiveness of this approach in selecting a subset of queries that forms a clustered workload.

To further explore reductions in index memory footprint, we first analyzed the factors influencing query latency in HNSW and whether improvements could be achieved through workload knowledge. We investigated methods to refine the selection of vectors in the upper layers, since the default HNSW index construction algorithm selects them uniformly at random. Our intuition was that areas of the search space with high query frequency should be over-represented with entry points, while sparsely accessed areas should be under-represented. We demonstrated that initializing searches at layer 0 from the top-1 result of a query (a best-case scenario reflecting this intuition) significantly reduced the number of distance computations for low *ef_Search* values. However, this benefit diminished rapidly as *ef_Search* increased, since higher values encourage broader graph exploration, hence diluting the advantage of starting closer to the query vector. A

similar trend was observed in recall. At low *ef_Search* values, searches are more prone to local minima, leading to greater variability and higher standard deviation in results. As *ef_Search* increases, the optimized and standard searches converge to nearly identical sets of nodes, thus producing equivalent recall.

We then proceeded to investigate techniques for reducing the memory footprint of HNSW using workload knowledge. We proposed strategies for assigning a *cache priority* to each node. Given a fixed *memory budget*, nodes with the highest priority are retained in memory, while the remainder are stored on disk. Our objective was to design a caching strategy that minimizes query latency for a clustered workload under fixed memory constraints. To this end, we proposed and evaluated multiple approaches: Most Frequently Used (MFU) (baseline), Expand from the Visited Set (EVS), Expand from the Visited Set In-Degree (EVSI), Personalized PageRank (PPR), and Heat Kernel PageRank (HKPR).

To generate the workload to evaluate these methods, we applied our clustered workload generation technique to ANN datasets. We used metrics such as the average in-memory hit rate and the proportion of queries with at least $x\%$ of accessed nodes already cached. In particular, $x = 100$ indicates queries that can be answered entirely from memory, which was a primary optimization goal. Our results show that across varied configurations of dataset size, *memory budget*, *ef_Search*, and workload structure, HKPR consistently outperforms or at least matches the performance of other methods. In terms of serving queries entirely from memory, HKPR demonstrated improvements of up to an order of magnitude over the baseline, EVS, and EVSI. Additionally, both PPR and especially HKPR exhibited superior generalization, requiring substantially fewer training queries to achieve strong performance, thereby reducing data collection costs.

We also examined the effect of ignoring nodes stored on disk and performing searches solely on the in-memory subgraph. We observed that while this strategy can incur recall loss for queries with a low fraction of in-memory vectors, queries with a high fraction cached exhibited negligible recall degradation. Since HKPR maintained a significantly larger fraction of queries with most of their accessed nodes cached compared to other methods, these queries could be served entirely from memory, with no disk I/O and minimal recall loss.

In all of the aforementioned experiments, we have always sampled 50% of the available workload queries to choose the subset of vectors to cache. While this is useful to showcase the potential of all approaches, it greatly favours the baseline, EVS and EVSI, as they have a smaller generalization ability. As showed in subsection 6.3.8, PPR and especially HKPR show little deterioration as the number of training queries decreases, as long as they remain representative of the workload. As an example, HKPR showed better results with a 2% training sample of the workload than all other methods (except PPR) with 30%. Additionally, for small training set sizes, HKPR and PPR perform orders of magnitude better than the other methods, due to their generalization ability. One takeaway is that EVS and EVSI only work properly when the training set size is comparable to the number of incoming queries. PPR and HKPR, on the other

hand, work well when the training set size is orders of magnitude smaller than the the incoming workload. For our application, this scenario is more realistic as we intend to use a small number of training queries to decide what nodes to cache for a much bigger workload. Additionally, requiring few training examples allows these methods to adapt quicker to shifting workloads.

Finally, we compared the number of disk accesses between our methods and DiskANN. Although our methods needs more vectors in memory overall — since all accessed vectors must reside in memory at some point, compared to only the search path vectors in DiskANN — our approaches outperformed DiskANN at larger *memory budget* values or with growing dataset sizes. This is because DiskANN caches a representative sample of the entire search space or nodes near entry points [24], whereas our methods exploit workload knowledge to prioritize vectors most likely to be visited in practice.

Future Work

Our study highlights the potential of workload-aware caching strategies based on random-walk techniques (especially PPR and HKPR), but several open challenges remain.

First, our methods assume a clustered query workload, yet we do not address how to detect clustering in practice or identify distributional shifts that may necessitate reassigning vectors to memory. This is an essential challenge for making these methods practical.

Second, we did not investigate dynamic cache updates. Future research could explore integrating our caching strategies with adaptive mechanisms to avoid full recomputation of cache priorities.

Third, while our experiments focused on clustered workloads, extending these strategies to other workload types — or to different graph-based indexes — remains unexplored. Similarly, measuring what is the performance on the aforementioned metrics of out-of-distribution queries is not assessed in the thesis and left as future work.

Our methods assume no particular disk storage layout, which improves generalizability, but incorporating layout-aware optimizations (such as *piggybacking* and prefetching) could further reduce the number of I/O operations.

Fourth, if computing HKPR or PPR proves too costly, approximate or sampling-based alternatives could make these methods more practical. We leave exploration of the impact of such approximations on performance as future work.

Fifth, we did not vary index construction parameters, leaving exploration of how such parameters influence cache effectiveness as an important area for further investigation. Heuristics to decide, on a per-query basis, whether to ignore disk nodes or not could prevent recall degradation for outlier queries while still reducing latency for typical workloads.

Finally, all evaluations in this thesis were conducted on generated workloads. Applying our methods to real-world workloads will be crucial to validate their practical utility.

Glossary

ANN Approximate Nearest Neighbor. 5, 8, 9, 14, 17, 18, 20, 21, 27, 33, 36, 40, 65, 66, 72, 116, 117

BFS Breadth-First Search. 14, 57, 58, 86

EVS Expand from the Visited Set. 6, 14, 15, 16, 17, 56, 59, 64, 65, 66, 67, 68, 71, 76, 77, 81, 83, 84, 90, 92, 93, 100, 109, 110, 111, 117

EVSI Expand from the Visited Set In-Degree. 6, 59, 64, 66, 67, 68, 71, 81, 83, 84, 90, 92, 93, 100, 109, 110, 111, 117

FIFO First In First Out. 58

HKPR Heat Kernel PageRank. 3, 4, 6, 14, 15, 16, 17, 62, 63, 64, 66, 67, 68, 76, 77, 80, 81, 83, 84, 86, 90, 92, 93, 97, 100, 101, 102, 103, 104, 106, 109, 110, 111, 114, 115, 117, 118

HNSW Hierarchical Navigable Small World. 3, 4, 5, 8, 9, 10, 11, 12, 13, 15, 16, 17, 22, 23, 24, 27, 28, 29, 30, 31, 34, 35, 37, 38, 40, 41, 42, 43, 46, 48, 50, 51, 52, 53, 56, 60, 61, 62, 63, 65, 67, 70, 73, 74, 86, 91, 105, 109, 110, 112, 113, 114, 116, 117

IVF Inverted File Index. 22

LFU Least Frequently Used. 10, 25

LLM Large Language Model. 3, 8, 56

LRU Least Recently Used. 10, 13, 25

LSH Locality-Sensitive Hashing. 21

MFU Most Frequently Used. 6, 13, 25, 54, 67, 109, 117

MRU Most Recently Used. 25

NSW Navigable Small World. 8, 9, 11, 12, 17, 23, 24, 38, 43, 46, 48, 50, 112

PPR Personalized PageRank. 6, 14, 15, 16, 59, 60, 61, 62, 63, 64, 66, 67, 68, 76, 77, 83, 84, 90, 92, 93, 97, 100, 101, 102, 103, 104, 109, 110, 114, 117, 118

PQ Product Quantization. 22, 113

RAG Retrieval Augmented Generation. 8, 19

RNG Relative Neighborhood Graph. 24, 43, 113

SQ Scalar Quantization. 22

VDBMS Vector Database Management System. 3, 5, 8, 19, 20, 25, 28, 40, 65, 92

Bibliography

[1] Cecilia Aguerrebere, Ishwar Bhati, Mark Hildebrand, Mariano Tepper, and Ted Willke. *Similarity search in the blink of an eye with compressed indices*. 2023. arXiv: 2304.04759 [cs.LG]. URL: https://arxiv.org/abs/2304.04759.

[2] Xingyan Bin, Jianfei Cui, Wujie Yan, Zhichen Zhao, Xintian Han, Chongyang Yan, Feng Zhang, Xun Zhou, Qi Wu, and Zuotao Liu. *Real-time Indexing for Large-scale Recommendation by Streaming Vector Quantization Retriever*. 2025. arXiv: 2501.08695 [cs.IR]. URL: https://arxiv.org/abs/2501.08695.

[3] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. "A Deeper Investigation of PageRank as a Function of the Damping Factor". In: *Web Information Retrieval and Linear Algebra Algorithms*. Ed. by Andreas Frommer, Michael W. Mahoney, and Daniel B. Szyld. Vol. 7071. Dagstuhl Seminar Proceedings (DagSemProc). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2007, pp. 1–19. DOI: 10.4230/DagSemProc.07071.3. URL: https://drops.dagstuhl.de/entities/document/10.4230/DagSemProc.07071.3.

[4] Tadeusz Caliński and Harabasz JA. "A Dendrite Method for Cluster Analysis". In: *Communications in Statistics - Theory and Methods* 3 (Jan. 1974), pp. 1–27. DOI: 10.1080/03610927408827101.

[5] Karthekeyan Chandrasekaran, Daniel Dadush, Venkata Gandikota, and Elena Grigorescu. *Lattice-based Locality Sensitive Hashing is Optimal*. 2017. arXiv: 1712.08558 [cs.DS]. URL: https://arxiv.org/abs/1712.08558.

[6] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. *SPANN: Highly-efficient Billion-scale Approximate Nearest Neighbor Search*. 2021. arXiv: 2111.08566 [cs.DB]. URL: https://arxiv.org/abs/2111.08566.

[7] Tobias Christiani. *Fast Locality-Sensitive Hashing Frameworks for Approximate Near Neighbor Search*. 2018. arXiv: 1708.07586 [cs.DS]. URL: https://arxiv.org/abs/1708.07586.

[8] Fan Chung. "The heat kernel as the pagerank of a graph". In: *Proceedings of the National Academy of Sciences* 104.50 (2007), pp. 19735–19740. DOI: 10.1073/pnas.0708838104. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.0708838104. URL: https://www.pnas.org/doi/abs/10.1073/pnas.0708838104.

[9] Fan Chung and Olivia Simpson. *Computing Heat Kernel Pagerank and a Local Clustering Algorithm*. 2016. arXiv: 1503.03155 [cs.DS]. URL: https://arxiv.org/abs/1503.03155.

[10] Fan Chung and Olivia Simpson. “Solving Local Linear Systems with Boundary Conditions Using Heat Kernel Pagerank”. In: *Internet Mathematics* 11.4–5 (Jan. 2015), pp. 449–471. ISSN: 1944-9488. DOI: 10.1080/15427951.2015.1009522. URL: http://dx.doi.org/10.1080/15427951.2015.1009522.

[11] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. *The Faiss library*. 2025. arXiv: 2401.08281 [cs.LG]. URL: https://arxiv.org/abs/2401.08281.

[12] E. B. Dynkin. “Kolmogorov and the Theory of Markov Processes”. In: *The Annals of Probability* 17.3 (1989), pp. 822–832. ISSN: 00911798, 2168894X. URL: http://www.jstor.org/stable/2244385 (visited on 07/18/2025).

[13] Yuval Eldar, Michael Lindenbaum, Moshe Porat, and Yehoshua Zeevi. “The Farthest Point Strategy for Progressive Image Sampling”. In: *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society* 6 (Feb. 1997), pp. 1305–15. DOI: 10.1109/83.623193.

[14] Owen P. Elliott and Jesse Clark. “The Impacts of Data, Ordering, and Intrinsic Dimensionality on Recall in Hierarchical Navigable Small Worlds”. In: *Proceedings of the 2024 ACM SIGIR International Conference on Theory of Information Retrieval*. ICTIR ’24. ACM, Aug. 2024, pp. 25–33. DOI: 10.1145/3664190.3672512. URL: http://dx.doi.org/10.1145/3664190.3672512.

[15] Bohao Feng, Hua-Chun Zhou, Guang-Lei Li, Hong-Ke Zhang, and Han-Chieh Chao. “Least Popularly Used: A Cache Replacement Policy for Information-Centric Networking”. In: 17.1 (Jan. 2016), pp. 1–10. ISSN: 1607-9264. DOI: 10.6138/JIT.2016.17.1.20151208.

[16] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. *Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph*. 2018. arXiv: 1707.00143 [cs.LG]. URL: https://arxiv.org/abs/1707.00143.

[17] David F. Gleich. *PageRank beyond the Web*. 2014. arXiv: 1407.5107 [cs.SI]. URL: https://arxiv.org/abs/1407.5107.

[18] Hyein Gu and {Min Soo} Kim. “ScaDANN: A Scalable Disk-Based Graph Indexing Method for ANN”. English. In: *CEUR Workshop Proceedings* 3946 (2025). Publisher Copyright: Copyright © 2025 for this paper by its authors.; Workshops of the EDBT/ICDT 2025 Joint Conference, EDBT/ICDT-WS 2025 ; Conference date: 25-03-2025. ISSN: 1613-0073.

[19] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[20] Taher H. Haveliwala. “Topic-sensitive PageRank”. In: *Proceedings of the 11th International Conference on World Wide Web*. WWW ’02. Honolulu, Hawaii, USA: Association for Computing Machinery, 2002, pp. 517–526. ISBN: 1581134495. DOI: 10.1145/511446.511513. URL: https://doi.org/10.1145/511446.511513.

[21] Indra Herdiana, M Alfin Kamal, Triyani, Mutia Nur Estri, and Renny. *A More Precise Elbow Method for Optimum K-means Clustering*. 2025. arXiv: 2502.00851 [stat.ME]. URL: https://arxiv.org/abs/2502.00851.

[22] Shikhar Jaiswal, Ravishankar Krishnaswamy, Ankit Garg, Harsha Vardhan Simhadri, and Sheshansh Agrawal. *OOD-DiskANN: Efficient and Scalable Graph ANNS for Out-of-Distribution Queries*. 2022. arXiv: 2211.12850 [cs.LG]. URL: https://arxiv.org/abs/2211.12850.

[23] Rajesh Jayaram, Erik Waingarten, and Tian Zhang. *Data-Dependent LSH for the Earth Mover’s Distance*. 2024. arXiv: 2403.05041 [cs.DS]. URL: https://arxiv.org/abs/2403.05041.

[24] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. “DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper_files/paper/2019/file/09853c7fb1d3f8ee67a61b6bf4a7f8e6-Paper.pdf.

[25] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. *Searching in one billion vectors: re-rank with source coding*. 2011. arXiv: 1102.3828 [cs.IR]. URL: https://arxiv.org/abs/1102.3828.

[26] Glen Jeh and Jennifer Widom. “Scaling personalized web search”. In: *Proceedings of the 12th International Conference on World Wide Web*. WWW ’03. Budapest, Hungary: Association for Computing Machinery, 2003, pp. 271–279. ISBN: 1581136803. DOI: 10.1145/775152.775191. URL: https://doi.org/10.1145/775152.775191.

[27] Xin Jin and Jiawei Han. “K-Means Clustering”. In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2010, pp. 563–564. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_425. URL: https://doi.org/10.1007/978-0-387-30164-8_425.

[28] Leo Katz. “A New Status Index Derived from Sociometric Analysis”. In: *Psychometrika* 18.1 (1953), pp. 39–43. DOI: 10.1007/BF02289026.

[29] Johannes Klicpera, Aleksandar Bojchevski, and Stephan Günnemann. “Predict then Propagate: Graph Neural Networks meet Personalized PageRank”. In: *International Conference on Learning Representations*. 2018. URL: https://api.semanticscholar.org/CorpusID:67855539.

[30] Kyle Kloster and David F. Gleich. *Heat kernel based community detection*. 2016. arXiv: 1403.3148 [cs.SI]. URL: https://arxiv.org/abs/1403.3148.

[31] Emmanouil Krasanakis, Symeon Papadopoulos, Ioannis Kompatsiaris, and Andreas Symeonidis. “pygrank: A Python Package for Graph Node Ranking”. In: *SoftwareX* (Oct. 2022). DOI: 10.1016/j.softx.2022.101227. URL: https://doi.org/10.1016/j.softx.2022.101227.

[32] Xin Liu and Kenneth Salem. “Hybrid storage management for database systems”. In: *Proc. VLDB Endow.* 6.8 (June 2013), pp. 541–552. ISSN: 2150-8097. DOI: 10.14778/2536354.2536355. URL: https://doi.org/10.14778/2536354.2536355.

[33] Peter Lofgren, Siddhartha Banerjee, and Ashish Goel. “Personalized PageRank Estimation and Search: A Bidirectional Approach”. In: *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*. WSDM ’16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 163–172. ISBN: 9781450337168. DOI: 10.1145/2835776.2835823. URL: https://doi.org/10.1145/2835776.2835823.

[34] Peter A. Lofgren, Siddhartha Banerjee, Ashish Goel, and C. Seshadhri. “FAST-PPR: scaling personalized pagerank estimation for large graphs”. In: *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’14. New York, New York, USA: Association for Computing Machinery, 2014, pp. 1436–1445. ISBN: 9781450329569. DOI: 10.1145/2623330.2623745. URL: https://doi.org/10.1145/2623330.2623745.

[35] Laurens van der Maaten and Geoffrey Hinton. “Visualizing Data using t-SNE”. In: *Journal of Machine Learning Research* 9.86 (2008), pp. 2579–2605. URL: http://jmlr.org/papers/v9/vandermaaten08a.html.

[36] Vasilis Mageirakos, Bowen Wu, and Gustavo Alonso. *Cracking Vector Search Indexes*. 2025. arXiv: 2503.01823 [cs.DB]. URL: https://arxiv.org/abs/2503.01823.

[37] Yu. A. Malkov and D. A. Yashunin. *Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs*. 2018. arXiv: 1603.09320 [cs.DS]. URL: https://arxiv.org/abs/1603.09320.

[38] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. “Approximate nearest neighbor algorithm based on navigable small world graphs”. In: *Information Systems* 45 (2014), pp. 61–68. ISSN: 0306-4379. DOI: https://doi.org/10.1016/j.is.2013.10.006. URL: https://www.sciencedirect.com/science/article/pii/S0306437913001300.

[39] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. “Scalable Distributed Algorithm for Approximate Nearest Neighbor Search Problem in High Dimensional General Metric Spaces”. In: *Similarity Search and Applications*. Ed. by Gonzalo Navarro and Vladimir Pestov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 132–147. ISBN: 978-3-642-32153-5.

[40] Dhruv Mátáni, Ketan Shah, and Anirban Mitra. “An O(1) algorithm for implementing the LFU cache eviction scheme”. In: *ArXiv* abs/2110.11602 (2021). URL: https://api.semanticscholar.org/CorpusID:239616461.

[41] Leland McInnes, John Healy, and James Melville. *UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction*. 2020. arXiv: 1802.03426 [stat.ML]. URL: https://arxiv.org/abs/1802.03426.

[42] Jason Mohoney, Anil Pacaci, Shihabur Rahman Chowdhury, Umar Farooq Minhas, Jeffery Pound, Cedric Renggli, Nima Reyhani, Ihab F. Ilyas, Theodoros Rekatsinas, and Shivaram Venkataraman. *Incremental IVF Index Maintenance for Streaming Vector Search*. 2024. arXiv: 2411.00970 [cs.DB]. URL: https://arxiv.org/abs/2411.00970.

[43] 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*. 2023. arXiv: 2304.01926 [cs.DB]. URL: https://arxiv.org/abs/2304.01926.

[44] Blaise Munyampirwa, Vihan Lakshman, and Benjamin Coleman. *Down with the Hierarchy: The 'H' in HNSW Stands for "Hubs"*. 2025. arXiv: 2412.01940 [cs.LG]. URL: https://arxiv.org/abs/2412.01940.

[45] Hy Nguyen, Nguyen Hung Nguyen, Nguyen Linh Bao Nguyen, Srikanth Thudumu, Hung Du, Rajesh Vasa, and Kon Mouzakis. *Dual-Branch HNSW Approach with Skip Bridges and LID-Driven Optimization*. 2025. arXiv: 2501.13992 [cs.LG]. URL: https://arxiv.org/abs/2501.13992.

[46] Jiongkang Ni, Xiaoliang Xu, Yuxiang Wang, Can Li, Jiajie Yao, Shihai Xiao, and Xuecang Zhang. *DiskANN++: Efficient Page-based Search over Isomorphic Mapped Graph Index using Query-sensitivity Entry Vertex*. 2023. arXiv: 2310.00402 [cs.IR]. URL: https://arxiv.org/abs/2310.00402.

[47] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. "The LRU-K page replacement algorithm for database disk buffering". In: *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*. SIGMOD '93. Washington, D.C., USA: Association for Computing Machinery, 1993, pp. 297–306. ISBN: 0897915925. DOI: 10.1145/170035.170081. URL: https://doi.org/10.1145/170035.170081.

[48] Aude Oliva and Antonio Torralba. "Modeling the Shape of the Scene: A Holistic Representation of the Spatial Envelope". In: *International Journal of Computer Vision* 42 (May 2001), pp. 145–175. DOI: 10.1023/A:1011139631724.

[49] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. *The PageRank Citation Ranking: Bringing Order to the Web*. Technical Report 1999-66. Previous number = SIDL-WP-1999-0120. Stanford InfoLab, Nov. 1999. URL: http://ilpubs.stanford.edu:8090/422/.

[50] 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*. SIGMOD/PODS '24. Santiago AA, Chile: Association for Computing Machinery, 2024, pp. 597–604. ISBN: 9798400704222. DOI: 10.1145/3626246.3654691. URL: https://doi.org/10.1145/3626246.3654691.

[51] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. “GloVe: Global Vectors for Word Representation”. In: *Empirical Methods in Natural Language Processing (EMNLP)*. 2014, pp. 1532–1543. URL: http://www.aclweb.org/anthology/D14-1162.

[52] P. Yogendra Prasad, Lakshmi Narayana Velayduam, Srirama Sai Kumar, Momin Shaikshavali, V. Vamsidhar Reddy, and Nayudori Ajay Kumar. “Selection of Cache Replacement Algorithm for PostgreSQL”. In: *2023 International Conference on Sustainable Computing and Data Communication Systems (ICSCDS)*. 2023, pp. 1715–1720. DOI: 10.1109/ICSCDS56580.2023.10105126.

[53] William Pugh. “Skip lists: a probabilistic alternative to balanced trees”. In: *Commun. ACM* 33.6 (June 1990), pp. 668–676. ISSN: 0001-0782. DOI: 10.1145/78973.78977. URL: https://doi.org/10.1145/78973.78977.

[54] Jie Ren, Minjia Zhang, and Dong Li. “HM-ANN: efficient billion-point nearest neighbor search on heterogeneous memory”. In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. NIPS ’20. Vancouver, BC, Canada: Curran Associates Inc., 2020. ISBN: 9781713829546.

[55] Peter J. Rousseeuw. “Silhouettes: A graphical aid to the interpretation and validation of cluster analysis”. In: *Journal of Computational and Applied Mathematics* 20 (1987), pp. 53–65. ISSN: 0377-0427. DOI: https://doi.org/10.1016/0377-0427(87)90125-7. URL: https://www.sciencedirect.com/science/article/pii/0377042787901257.

[56] Urvi Sharma, G P Sajeev, and S Siji Rani. “Personalized Fashion Recommendation Using Nearest Neighbor PageRank Algorithm”. In: *2022 International Conference on Connected Systems and Intelligence (CSI)*. 2022, pp. 1–6. DOI: 10.1109/CSI54720.2022.9924114.

[57] Harsha Vardhan Simhadri, Ravishankar Krishnaswamy, Gopal Srinivasa, Suhas Jayaram Subramanya, Andrija Antonijevic, Dax Pryce, David Kaczynski, Shane Williams, Siddarth Gollapudi, Varun Sivashankar, Neel Karia, Aditi Singh, Shikhar Jaiswal, Neelam Mahapatro, Philip Adams, Bryan Tower, and Yash Patel. *DiskANN: Graph-structured Indices for Scalable, Fast, Fresh and Filtered Approximate Nearest Neighbor Search*. Version 0.6.1. 2023. URL: https://github.com/Microsoft/DiskANN.

[58] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. *FreshDiskANN: A Fast and Accurate Graph-Based ANN Index for Streaming Similarity Search*. 2021. arXiv: 2105.09613 [cs.IR]. URL: https://arxiv.org/abs/2105.09613.

[59] Sivic and Zisserman. “Video Google: a text retrieval approach to object matching in videos”. In: *Proceedings Ninth IEEE International Conference on Computer Vision*. 2003, 1470–1477 vol.2. DOI: 10.1109/ICCV.2003.1238663.

[60] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnaswamy, and Rohan Kadekodi. “Rand-NSG: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node”. In: *Neural Information Processing Systems*. 2019. URL: https://api.semanticscholar.org/CorpusID:202770414.

[61] Ghaidaa A. Al-Sultany. “Enhancing Recommendation System using Adapted Personalized PageRank Algorithm”. In: *2022 5th International Conference on Engineering Technology and its Applications (IICETA)*. 2022, pp. 1–5. DOI: 10.1109/IICETA54559.2022.9888678.

[62] Toni Taipalus. “Vector database management systems: Fundamental concepts, use-cases, and current challenges”. In: *Cognitive Systems Research* 85 (2024), p. 101216. ISSN: 1389-0417. DOI: https://doi.org/10.1016/j.cogsys.2024.101216. URL: https://www.sciencedirect.com/science/article/pii/S1389041724000093.

[63] Kento Tatsuno, Daisuke Miyashita, Taiga Ikeda, Kiyoshi Ishiyama, Kazunari Sumiyoshi, and Jun Deguchi. *AiSAQ: All-in-Storage ANNS with Product Quantization for DRAM-free Information Retrieval*. 2025. arXiv: 2404.06004 [cs.IR]. URL: https://arxiv.org/abs/2404.06004.

[64] Yao Tian, Xi Zhao, and Xiaofang Zhou. *DB-LSH: Locality-Sensitive Hashing with Query-based Dynamic Bucketing*. 2022. arXiv: 2207.07823 [cs.DB]. URL: https://arxiv.org/abs/2207.07823.

[65] Robert Tibshirani, Guenther Walther, and Trevor Hastie. “Estimating the number of clusters in a data set via the gap statistic”. In: *Journal of the Royal Statistical Society Series B* 63.2 (2001), pp. 411–423. URL: https://EconPapers.repec.org/RePEc:bla:jorssb:v:63:y:2001:i:2:p:411-423.

[66] Godfried T. Toussaint. “The relative neighbourhood graph of a finite planar set”. In: *Pattern Recognition* 12.4 (1980), pp. 261–268. ISSN: 0031-3203. DOI: https://doi.org/10.1016/0031-3203(80)90066-7. URL: https://www.sciencedirect.com/science/article/pii/0031320380900667.

[67] Bhisham Dev Verma and Rameshwar Pratap. *Faster and Space Efficient Indexing for Locality Sensitive Hashing*. 2025. arXiv: 2503.06737 [cs.DS]. URL: https://arxiv.org/abs/2503.06737.

[68] Daniel Vial and Vijay Subramanian. “A Structural Result for Personalized PageRank and its Algorithmic Consequences”. In: *Proc. ACM Meas. Anal. Comput. Syst.* 3.2 (June 2019). DOI: 10.1145/3341617.3326140. URL: https://doi.org/10.1145/3341617.3326140.

[69] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stefan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antonio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[70] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. *A Comprehensive Survey and Experimental Comparison of Graph-Based Approximate Nearest Neighbor Search*. 2021. arXiv: 2101.12631 [cs.IR]. URL: https://arxiv.org/abs/2101.12631.

[71] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, Peng Cheng, and Mao Yang. "SPFresh: Incremental In-Place Update for Billion-Scale Vector Search". In: *Proceedings of the 29th Symposium on Operating Systems Principles*. SOSP '23. ACM, Oct. 2023, pp. 545–561. DOI: 10.1145/3600006.3613166. URL: http://dx.doi.org/10.1145/3600006.3613166.

[72] Gulshan Yadav, RahulKumar Yadav, Mansi Viramgama, Mayank Viramgama, and Apeksha Mohite. *Quantixar: High-performance Vector Data Management System*. 2024. arXiv: 2403.12583 [cs.DB]. URL: https://arxiv.org/abs/2403.12583.

[73] Renchi Yang, Xiaokui Xiao, Zhewei Wei, Sourav S. Bhowmick, Jun Zhao, and Rong-Hua Li. "Efficient Estimation of Heat Kernel PageRank for Local Clustering". In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD '19. Amsterdam, Netherlands: Association for Computing Machinery, 2019, pp. 1339–1356. ISBN: 9781450356435. DOI: 10.1145/3299869.3319886. URL: https://doi.org/10.1145/3299869.3319886.

[74] Cheng Zhang, Jianzhi Wang, Wan-Lei Zhao, and Shihai Xiao. "Highly Efficient Disk-based Nearest Neighbor Search on Extended Neighborhood Graph". In: *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '25. Padua, Italy: Association for Computing Machinery, 2025, pp. 2513–2523. ISBN: 9798400715921. DOI: 10.1145/3726302.3729996. URL: https://doi.org/10.1145/3726302.3729996.

[75] Yinglong Zhang, Xuewen Xia, Xing Xu, Fei Yu, Hongrun Wu, Ying Yu, and Bo Wei. "Robust Hierarchical Overlapping Community Detection With Personalized PageRank". In: *IEEE Access* 8 (2020), pp. 102867–102882. DOI: 10.1109/ACCESS.2020.2998860.

[76] Shurui Zhong, Dingheng Mo, and Siqiang Luo. *LSM-VEC: A Large-Scale Disk-Based System for Dynamic Vector Search*. 2025. arXiv: 2505.17152 [cs.DB]. URL: https://arxiv.org/abs/2505.17152.