SPIN: CACHE-EFFICIENT GRAPH NEAREST NEIGHBOR SEARCH

Anonymous authors
Paper under double-blind review

ABSTRACT

Approximate nearest-neighbor (ANN) search is widely employed in numerous fields that require combing through large databases for relevant information in response to a particular query. Graph-based algorithms, such as Hierarchical Navigable Small World (HNSW), have become dominant tools for ANN searches. However, query costs in these graphs are dominated by irregular memory accesses, which cause frequent cache misses, leading to higher latency and lower throughput. Coleman et al. (Neurips 2022) remedies this by optimizing the memory layout of nodes while keeping the edge set intact. We ask whether the graph's structural properties provide an orthogonal mechanism for addressing this issue. The bottom layer of a dense HNSW graph exhibits heavy inherent degree skew, and because of its hierarchical nature, many edges in this layer are redundant and rarely traversed. We exploit this by applying a geometry-based sparsifier that prunes the bottom-layer edges following a fixed budget while retaining each node's geometrically nearest neighbors with high probability and preserving a close-to-dense recall. The sparsified index is then partitioned and reordered in memory, ensuring that co-visited nodes are placed in contiguous memory blocks. Across scales, the combined pipeline provides a formal guarantee that scaling the search effort by a constant returns a close-to-dense result with high probability. When combined with the sparsified cache-aware layout, the method reduces memory traffic and improves performance.

1 INTRODUCTION & RELATED WORK

Graph-based nearest neighbor algorithms have emerged as a popular tool for searching large vector databases and form a key component of modern retrieval-augmented generation (RAG) systems (Fan et al., 2024). In a typical RAG pipeline, a collection of documents is first divided into passages or chunks and mapped into a high-dimensional vector space using an embedding model. At inference time, the user's query is embedded into the same space, and a nearest-neighbor search is performed to identify the database embeddings most similar to the query. The corresponding passages are then supplied to a large language model (LLM) as additional context, allowing the model to generate a response grounded in information retrieved from the underlying corpus (Lewis et al., 2020). Since vector databases used in RAG applications can contain millions of embeddings, performing an exhaustive comparison against every stored vector is prohibitively expensive (Johnson et al., 2021). Consequently, practical RAG systems commonly rely on approximate nearest neighbor (ANN) indexes such as graph-based constructions (Zhu et al., 2026). Notable examples of graph-based vector indexes that are widely adopted in real-world applications include small-world architectures (Malkov et al., 2014) and HNSW (Malkov & Yashunin, 2020), DiskANN (Subramanya et al., 2019), and models like FANNG (Harwood & Drummond, 2016). Compared to traditional nearest-neighbor search methods, such as locality-sensitive hashing (Datar et al., 2004), these constructions feature better search parameters and navigability heuristics, achieving faster convergence and high recall (Ponomarenko, 2015; Li et al., 2026; Xie et al., 2025; Ruan et al., 2025; Conway et al., 2026; Yan & Wang, 2025; Wang et al., 2021; Azizi et al., 2025).

However, the intensive use of graph-based vector indexes necessitates that the search functionality operates in a high-throughput and low-latency regime, since every fetched passage directly adds to query latency of the overall system (Lewis et al., 2020; Karpukhin et al., 2020). Graph traversal is inherently an expensive process and the search time depends on: (i) the cost of fetching the relevant

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

portions of the graph from secondary storage, and (ii) the cost of accessing and traversing graph data once it resides in main memory. These performance concerns have led to the development of *system-aware* designs that focus on disk, I/O and memory efficiency(Subramanya et al., 2019; Wang et al., 2024; Chen et al., 2021; Ming et al., 2025), leverage GPU infrastructures (Kim et al., 2025; Li et al., 2025; Venkatasubba et al., 2025), parallel computing frameworks (Manohar et al., 2024; Wang et al., 2025; Muppuri, 2026) and distributed environments (Akhil & G, 2025; Deng et al., 2019; Guo & Lu, 2026), and utilize more performant search methods (Wang et al., 2023; Zheng et al., 2026).

An often overlooked performance bottleneck for graph-based index search is cache efficiency, a critical metric for high-throughput systems. Accessing data from the cache is still orders of magnitude faster than accessing the main memory on modern systems. Unfortunately, graphs are inherently not cache-friendly because, depending on their structure, they exhibit poor spatial and temporal locality. These features are important for better cache utilization and effective prefetching. Prior work on graph analytics shows that preprocessing the graph to produce more cache-efficient layouts results in significantly better performance (Wei et al., 2016; Arai et al., 2016; Chen & Chung, 2021). This motivates a similar line of inquiry for graph-based vector indexes.

The most relevant work addressing this question is by (Coleman et al., 2022). They show that cache-efficiency can be improved by reordering the graph with the objective of colocating nodes in memory that are frequently visited together; this significantly reduces cache misses during search. Since finding an optimal graph ordering is NP-hard, they approximate the objective using *query profiling* data that captures co-access patterns observed during representative searches. The resulting layout increases the likelihood that, when one node is accessed, other nodes likely to be visited soon are located in the same or nearby cache lines and can therefore benefit from spatial locality and hardware prefetching. Motivated by this line of research, we ask the following question: *Can we design nearest neighbor graph algorithms where both search accuracy and cache efficiency is theoretically quantifiable in terms of data?*

1.1 Our Contributions

We propose SPIN (**S**parsification and **P**artition-aware **IN**dexing), a general method for transforming a hierarchical graph-based ANN index, such as HNSW (Malkov & Yashunin, 2020), into a sparse, memory-friendly representation that (i) preserves the search result of the original graph or returns a better candidate with high probability, and (ii) reduces the cache cost of in-memory search. Our approach targets the bottom layer of the hierarchy, where most of the graph traversal is performed, while leaving the upper layers unchanged. We summarize the major contributions below.

- **Geometry-aware graph transformation.** We characterize cache efficiency through two complementary properties: *graph working-set size*, capturing the amount of graph data accessed during a query, and *spatial locality*, capturing how well temporally related accesses are colocated in memory. We develop SPIN to jointly optimize these two properties. First, a geometry-aware graph sparsification mechanism preferentially removes long edges, reducing adjacency-list sizes and the graph's working set while preserving local connectivity, which is important for nearest-neighbor search. Second, cluster-aware graph reordering co-locates similar nodes while constraining cluster working sets to fit within the LLC, improving cache reuse.
- **Theoretical guarantees.** We provide theoretical guarantees connecting graph sparsification, search quality, and cache efficiency. We show that geometry-aware sparsification induces bounded expected graph stretch and derive the parameter regime to preserve search recall with high probability. We further derive asymptotic bounds on the cache misses incurred by SPIN under the ideal-cache model. *A key finding of our (empirically validated) analysis shows that the tradeoff between search accuracy and cache performance on a graph transformed by* SPIN *can be tuned by a single parameter, independent of the query distribution.*
- **Experiments.** We evaluate SPIN against the native HNSW index and two state-of-the-art cache-aware graph reordering mechanisms (GORDER (Wei et al., 2016), PORDER (Coleman et al., 2022)) on MS MARCO, HotpotQA, and Natural Questions at three scales $(100K - 1M)$. Experiments show that at matched ef, SPIN produces a 30–36% gain in performance by reducing cache/LLC misses while achieving a near-identical recall.

2 PROBLEM DEFINITION

We focus our study on the **Hierarchical Navigable Small World** (HNSW) graph construction Malkov & Yashunin (2020) which has found application in many real world systems, most notably **FAISS** (Meta AI Research), **Elasticsearch** (Elastic), **Redis** (Redis), **MongoDB** (MongoDB), and to support the backend for AWS's RAG service, **Bedrock** (Amazon Web Services, 2025). This also provides a comparable baseline against relevant prior work (Coleman et al., 2022). An HNSW index (Malkov & Yashunin, 2020) organizes a given dataset into a hierarchy of proximity graphs where every point appears in the bottom layer (layer 0), while each upper layer contains an exponentially random subset. The higher layers have long-range connections to reach the query-appropriate graph region, and the lower layers provide more fine-grained connections to identify nearest neighbors.

Our method exclusively focuses on the last (dense) layer, and we formally define it as $G = (V, E)$. Each edge $e = \{u, v\} \in E$, has an associated edge weight $w_e := d(u, v)$. Queries are answered using *greedy best-first traversal*, starting in the top layer, and iteratively finding the closest neighbor to the query while descending down the hierarchy. Search quality is maintained using a single parameter, ef, which denotes the size of the *candidate set* of nodes to be explored. The candidate set is a priority queue, and if the closest candidate is farther from the query than the ones that have already been discovered, then search is terminated, and the top k results are returned. This graph traversal can incur high memory access costs because neighboring nodes may reside in different memory blocks, leading to frequent cache misses. *Our goal is to reduce cache misses during the process while retaining high search accuracy.* We formally define our objective below.

Cache-efficiency objective: Under the idealized cache model of Frigo et al. (1999) with a cache with block size B and capacity C and a fixed memory layout, the expected number of cache misses encountered while answering a query q can be split into two parts: 1) How many edges/nodes are visited during the search, and 2) how expensive each visit is due to cache misses.

Definition 1. *For a given NN graph G, and under a fixed query distribution $\mathcal{Q}$, the expected number of cache misses while serving a query $q \sim \mathcal{Q}$ is*

$$\mathbb{E}_{q\sim\mathcal{Q}}\big[M(G,q)\big] \;\approx\; \underbrace{\mathbb{E}_{q\sim\mathcal{Q}}\big[\textsc{WorkingSet}(G,q)\big]}_{\text{graph-structure term}} \times \underbrace{\frac{1}{\textsc{Locality}(B,C,P(G))}}_{\text{layout term}},$$

where $\textsc{WorkingSet}(\cdot)$ is the total memory footprint when serving a query q using the graph G, and $\textsc{Locality}(\cdot)$ quantifies the amount of data relevant to the search for a query that can be prefetched at the same time to the cache with a single cache miss. A high locality value implies fewer cache misses overall. Moreover, minimizing WORKINGSET and maximizing LOCALITY will optimize $\mathbb{E}_{q\sim\mathcal{Q}}\big[M(G,q)\big]$. Intuitively, the total cost results from two complementary effects: the working set size can be optimized by improving the graph structure, *oblivious to hardware specifications*, while locality depends on the available cache size and the graph's memory layout. SPIN optimizes both, as we will describe in Sec. 3. We present each individual objective below.

2.1 SPARSIFICATION OBJECTIVE

SPIN optimizes the working set size of a query, by producing a transform of the graph $\widetilde{G}$ such that $\mathbb{E}_{q\sim\mathcal{Q}}\big[\textsc{WorkingSet}(\widetilde{G},q)\big] \;\le\; \mathbb{E}_{q\sim\mathcal{Q}}\big[\textsc{WorkingSet}(G,q)\big]$. This is achieved through a *geometry-aware sparsification*. Specifically, we define the following sparsification objective.

Definition 2 (Sparsification Objective). *Given a NN graph $G = (V, E)$ built with the distance metric $d(\cdot,\cdot)$ and search algorithm $\mathcal{A}(\cdot,\cdot)$. For a fixed query distribution $\mathcal{Q}$, fix a* sparsification parameter $\alpha \in (0, 1]$ *define a sparse graph $\widetilde{G} = (V, \widetilde{E})$ with $\widetilde{E} \subseteq E : |\widetilde{E}| \lesssim (1-\alpha)|E|$. Then, our objective is*

$$\arg\min_{\alpha\in(0,1]} \mathbb{E}_{q\sim\mathcal{Q}}\big[\textsc{WorkingSet}(\widetilde{G},q)\big] \quad \text{s.t.}$$
$$\mathbb{P}\Big[d(\mathcal{A}(\widetilde{G},q),q) \le d(\mathcal{A}(G,q),q)\Big] \ge 1-\delta$$

where δ is a small constant.

Intuitively, the objective tries to find the value of α such that the expected working set size during a query is minimized, but the search quality is retained, i.e., for a given query, the search algorithm returns either the same or a better result than the one found in the dense graph. Note that random sparsification is not an optimal choice, since it cannot ensure that the sparse graph has similar reachability values as the dense graph, and is also unlikely to yield favorable search quality bounds.

2.2 Cache Locality Objective

We state our cache locality objective in the ideal cache model for a fully associative cache of size $cachesize$ (Frigo et al., 1999). First, we define an affinity parameter for a pair of nodes, u and v. Intuitively, it measures how likely it is that the two nodes will be visited by a search during a query within $\approx (t - s)$ computation steps of each other, where $s < t$. Formally,

Definition 3. *For a graph $G = (V, E)$, and a query distribution $\mathcal{Q}$, define $\rho_{u,v}$ as a pairwise affinity measure*

$$\rho_{u,v} = \mathbb{E}_{q\sim\mathcal{Q}}\left[\sum_{1\le s<t\le|P_q|} g(t-s)\mathbf{1}\left\{\mathsf{visited}[s] = u,\ \mathsf{visited}[t] = v\right\}\right],$$

where g is a non-increasing function, and $\mathsf{visited}[t]$ denotes the node visited at search step t.

Definition 4 (Cache-capacity aware reordering). *For a graph $G = (V, E)$, and a query distribution $\mathcal{Q}$, let $\pi : V \to [1, \ldots, |V|]$ be a permutation of the nodes. Let a_u define the total amount of information (in bits) required to represent the node and its adjacency list. Let $\widetilde{\rho_{u,v}} := \rho_{u,v} + \rho_{v,u}$. Then, for a fixed size cache of C bits, our objective is: $\min_\pi \sum_{u,v\in V} \widetilde{\rho_{u,v}} \frac{|a_\pi(u)-a_\pi(v)|}{C}$.*

Intuitively, minimizing the objective ensures that nodes with high pairwise affinity over the query distribution are placed close to each other in the reordered layout, so they can fit into the cache together with their adjacency information. Thus, computation steps over these nodes during a search can be performed without incurring the overhead of a cache miss. The objective is a proxy for the locality term in our characterization of the expected cache miss. Consider that during a query q, the nodes visited are $v_1, \ldots, v_{|P_q|}$. If the nodes in the sequence $v_i, \ldots, v_j$, for $i < j$, have high pairwise affinity, then the objective will try to place them in proximity, and they will be prefetched to the cache. As such, $(j - i)$ steps of the search can be performed with a single cache miss, incurred when the initial block of data is fetched. Coleman et al. (2022) uses a similar objective and uses a set of profiling queries to learn a permutation. In SPIN, our goal is to solve the objective without using any additional profiling data.

3 Our Method: SPIN

SPIN follows a two-step process: i) produce a sparse graph for a fixed sparsification parameter, and ii) reorder the nodes of the sparse graph to reduce cache misses. We discuss these steps below.

3.1 Sparsification

Given a dense HNSW graph, our goal is to produce a smaller, sparsified graph, by deleting a certain fraction of the edges. However, NN graphs like HNSW have typically skewed constructions and selective edge importance. This makes the process more challenging than uniform edge removal.

- **Skewed hubness.** A small portion of the nodes in HNSW act as sources and sinks of a disproportionate number of edges, making the node degrees highly skewed. This skewness is a byproduct of HNSW's neighbor-selection RNG-pruning heuristics (see Table 3 in App. A).
- **Uneven edge importance.** HNSW has both long-range edges and short-range edges between similar nodes. However, once the search reaches the appropriate region, the long-range edges are rarely traversed: Table 4 (App. A) bins level-0 edges by length and shows that the longest 30% of edges are traversed at roughly $0.4\times$ the rate of the shortest 30% and account for under 19% of all traversal, with a negative rank correlation between edge length and traversal frequency ($\rho = -0.30, -0.22, -0.18$ at 100k/500k/1m). Removing short-range edges with the same importance as long-range edges is therefore bound to degrade search quality.

Distance-biased sparsification: These structural observations motivate our sparsification method: we use a *distance-biased* edge sampling for removing edges in the last layer of the graph, while keeping the rest of the layers intact. Specifically, given a sparsification parameter α, the probability of deleting an edge $e \in E$ is set to $p_e = \lambda w_e$ where the "edge weight" w_e is the normalized metric distance between the embeddings stored at u and v. Since we treat α as a fixed parameter, we solve for the value of λ under the constraint that for each edge $p_e = \min(1, \lambda w_e)$ and $1/|E| \sum_{e \in E} p_e) = \alpha$. Intuitively, edges between dissimilar nodes are prefered for deletion over edges that connect similar nodes with low metric distance. The rationale is that the hierarchical structure already accounts for longer edges in the upper layers, and it is less likely that the during a search in the last layer, a dissimilar node connected with a longer edge will be visited from the candidate set of the node under consideration.

The sparsification directly impacts two important parameters: i) reachability, and ii) increased path length between nodes. Ideally, the reachability after sparsification should be exactly the same as the dense graph, however, this requirement may be too restrictive and only allow very small levels of sparsification. Thus, we use a more flexible bound: if the reachable set in the dense graph from a vertex $v \in V$ is defined as $\mathcal{R}_G(v)$, and similarly the reachable set in the sparse graph for the same node is $\mathcal{R}_{\widetilde{G}}(v)$, then $\frac{\sum_{v \in V} |\mathcal{R}_{\widetilde{G}}(v)|}{\sum_{v \in V} |\mathcal{R}_G|} \geq 1 - \epsilon$ for all graph sizes up to a maximum size, and ϵ is a small value (empirically measured to be in the order of $\leq 10^{-3}$). Under this assumption, we bound the expected path length increase in App. G. Formally, we show that

Theorem 3.1 (**Expected multiplicative stretch (informal)**). *Given a graph $G = (V, E)$ and a graph $\widetilde{G} = (V, \widetilde{E})$ produced by sparsifying G with sparsification parameter α, the expected multiplicative stretch, defined as the expected increase in path length between two nodes due to sparsification is $\theta := \mathbb{E}\left[\mathcal{S}(G, \widetilde{G})\right] \in \mathrm{O}(\alpha + \beta)$ where β accounts for pairs of nodes where no constant-sized path exists after sparsification.*

Search accuracy: In practice, search accuracy will be defined by the recall for the nearest neighbor search, or based on the objective for a downstream task. However, analyzing recall as part of a general framework is problematic without making assumptions about the ground-truth data. Thus, for our analysis, we use a proxy term, namely the **response quality**: *what is the probability that the search on the sparse graph return the same result as the search on the dense graph, or a search candidate that is better?* A better search candidate is defined as an embedding that is closer under the metric distance to the query than the candidate returned by the search on the dense graph.

Naturally, the increase in path length will affect the response quality in the sparse graph under fixed search parameters. This can be compensated for by increasing the size of the candidate set explored during a search. More specifically, if the candidate set size for the dense graph is controlled by the parameter ef, and the multiplicative increase in path length between two nodes due to the sparsification is the parameter $\mathcal{S}(G, \widetilde{G})$, then we can set the candidate set size in the sparse graph as $\mathbb{E}\left[\mathcal{S}(G, \widetilde{G})\right] \cdot ef$ where the expectation is taken over all pairs of nodes. As we will show, this allows us to bound the response quality probabilistically with a Markov-like tail bound.

However, an important caveat here is that an expanded candidate set size $\mathbb{E}\left[\mathcal{S}(G, \widetilde{G})\right] \cdot ef$ does not immediately provide a useful bound because edge traversal during search is not uniformly distributed. Instead, there may be a small subset of edges with large increases in path length that are more frequently visited (due to the hubness property), while the expected value remains small. Thus, we must first estimate an effective edge-utilization measure for each edge using the query distribution, and then use this term to estimate the *weighted* expected increase in path length. In practice, we can estimate edge utilization using a set of representative queries. Formally, we prove in App. G

Theorem 3.2 (**Search accuracy (informal)**). *Given a query distribution $\mathcal{Q}$, let*

$$\chi = \frac{\textit{average path length stretch weighted by edge utilization in } \mathcal{Q}}{\textit{average path length stretch assuming uniform edge utilization}}$$

Let the candidate set size for exploration in the sparse graph be set to $\gamma \cdot ef$, where ef is the candidate set size for search in the dense graph, and $\gamma \in \left[1, \mathbb{E}\left[\mathcal{S}(G, \widetilde{G}(\alpha))\right]\right]$ is a fixed parameter.

Also, define a non-decreasing function $\delta \in [1, \gamma_{\max}] \to (1, \infty)$. Then, the probability that the sparse graph search returns a nearest neighbor candidate for query q, $\widetilde{x}_q^{(\gamma)}$, and this candidate is either the same or better than the candidate x_q returned by the search on the dense graph is bounded is

$$\mathbb{P}\left[d(q, \widetilde{x}_q^{(\gamma)}) \leq d(q, x_q)\right] \geq 1 - \rho_\gamma - \mathrm{O}\left(\frac{\chi \cdot \mathbb{E}\Big[\mathcal{S}(G, \widetilde{G}) - 1\Big]}{\delta(\gamma)}\right)$$

The probability ρ_γ accounts for the unlikely case where there is a short path to the candidate x_q in the sparsified graph (and x_q can be found during the search by setting the candidate set size γef), but the search instead explores a sub-optimal path and does not return x_q.

Implication: The result shows that search fails to return the same (or better) candidate when either the increased candidate set size is not sufficient to counteract the graph stretch due to sparsification, or the search returns a sub-optimal result despite proper parameter selection. We empirically show in Sec. 4.1 that these events are unlikely, and we achieve almost identical recall to search on the dense graph after sparsification when $\gamma = \mathbb{E}\Big[\mathcal{S}(G, \widetilde{G})\Big]$, and the recall drops gracefully as $\gamma \to 1$.

3.2 REORDERING

Graph reordering allows us to maximize the locality term in our objective, and combined with the reduction in average working set, this improves cache efficiency. Similar to (Coleman et al., 2022) , our objective is to locate nodes that are frequently visited together during search in the vicinity of each other. However, instead of using profiling queries to identify such nodes, we hypothesize that *the last layer search will likely explore nodes with high degree of similarity*. Intuitively, this is because upper-layer searches already explore larger hops with dissimilar nodes.

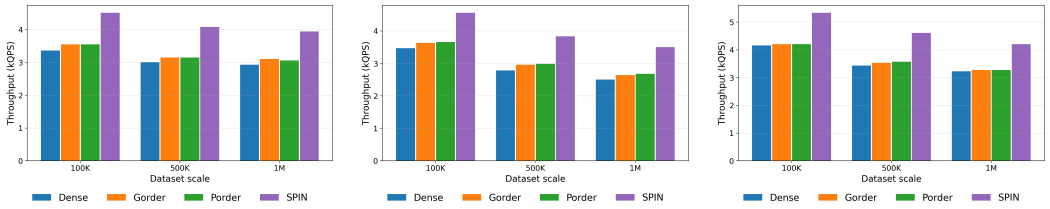
Based on this hypothesis, SPIN first clusters the nodes by similarity. *Crucially, the smaller edge lists enable us to perform this clustering for large graphs*. Moreover, the smaller working set size means that the information per cluster, i.e., the adjacency lists of all nodes, is small. We use METIS (Karypis & Kumar, 1998) to partition the nodes into K clusters, under the constraint that each cluster of nodes (including their adjacency lists) can fit inside the cache. After METIS assigns a cluster label to each node $\pi(v) \in \{1, \ldots, K\}$, the reorderer produces a new global order P_{opt} that places every node in the same partition contiguously in memory, each block roughly the size of the cache. Then the index is rebuilt following that order, without altering the graph topology.

For our cache performance analysis, we first note that estimating the correlations between node cluster label assignments is a good indicator of cluster quality, but it does not provide a reasonable metric for cache misses. Two nodes that are clustered together may be visited at different times during a search, and as such, if the sequence is long enough between their visitation times, the cluster and its constituent information may be evicted from the cache between the accesses.

Instead, a more appropriate metric is *rate of cross cluster transitions*, denoted by the ratio $\frac{T_q^{(\gamma)}}{|P_q^{(\gamma)}|}$, where $T_q^{(\gamma)}$ is defined as the number of times a new cluster has to be fetched to the cache in the sequence of node explorations to answer the query q in the sparse graph, with the candidate set size set to γef. $|P_q^{(\gamma)}|$ is the total number of nodes explored during the search.

Theorem 3.3 (Cache misses (informal)). *For a given query distribution $\mathcal{Q}$, the expected number of cache misses when searching the sparse, reorded graph produced by* SPIN*, $M_{\widetilde{G}} \in \mathrm{O}\left(1 + \frac{\mathbb{E}_{q \in \mathcal{Q}}[T_q^{(\gamma)}]}{\mathbb{E}_{q \in \mathcal{Q}}[|P_q^{(\gamma)}|]} \cdot \mathcal{H}_{\widetilde{G}, \mathcal{Q}}(\gamma ef)\right)$, where $\mathcal{H}_{\widetilde{G}, \mathcal{Q}}(\gamma ef)$ is the expected number of nodes visited during a search for a query from $\mathcal{Q}$ with candidate set size γef.*

Implication: It is intuitively clear from the description that $\mathcal{H}_{\widetilde{G}, \mathcal{Q}}(\gamma ef)$ is a non-decreasing function of $\gamma \in [1, \gamma_{\max}]$. Thus, the result shows that even when the cross-cluster transition rate remains constant, increasing γ increases cache misses. This is also empirically demonstrated in Sec. 4.2. *Importantly, we observe that the single parameter γ presents a clean tradeoff between better cache performance and higher search accuracy (Theorem 3.2).* This allows us to tune the parameter as per requirements, and we empirically demonstrate this trade-off in Sec. 4.2.



(a) MS MARCO (b) HotpotQA (c) NQ

Figure 1: Query throughput (kQPS, higher is better) at $ef = 128$ across MS MARCO, HotpotQA, and NQ. SPIN outperforms all the baselines in all settings.

4 EXPERIMENTS

Datasets and Embeddings. We evaluate SPIN on MS Marco passage (Nguyen et al., 2016), Natural Questions (NQ) (Kwiatkowski et al., 2019) and HotpotQA (Yang et al., 2018), across three scales (100k,500k, and 1M) embedded with `msmarco-distilbert-dot-v5` into d=768-dimensional embeddings. For MS MARCO we index the raw dot-product embeddings; for NQ and HotpotQA we use $\ell_2$normalization. All HNSW graphs use M=32 (base-layer capacity M_0=2M=64) and are built using FAISS (Douze et al., 2026) library. For each dataset, 1000 test queries and a disjoint profiling split of training queries are embedded in a similar way.

Baselines. We compare with the following baselines. *(i) Retrieval-graph baselines.* The dense HNSW index which is the unpruned graph (M_0=64) in native construction order. *(ii) Memory-layout baselines.* We compare our partition-guided reordering against the cache-locality reordering methods, each applied to the dense index: **Gorder** (Wei et al., 2016), and **Porder** (Coleman et al., 2022), a query-profiling-based ordering that weights the Gorder objective by traversal frequency.

Metrics. To evaluate our method in a cache-efficiency axis, we report query **throughput (kQPS)**, and, from the single `perf` event group, the **LLC load-miss rate** (`LLC-load-misses/LLC-loads`) as the direct residency measure, alongside the cache (L1) load-miss rate. Retrieval quality is measured by **Recall@**k ($k \in \{1, 10, 100\}$) against exact top-100 ground truth; we report Recall@10 as representative, since all k follow the same pattern. We also report two machine-independent performance metrics: the number of *distance computations* per query (`ndis`) and the number of expanded nodes, or *hops* (`nhops`). Since expanding a node requires retrieving its neighbor list and corresponding vectors, these quantities also serve as a measurement of memory footprint throughout the search.

Other Settings. *For our experiments, we set the candidate set size multiplier $\gamma = 1$, such that the ef value for the sparse graph search matches the ef value in the dense graph. Later in Sec. 4.2, we provide throughput and recall trends for other values of $\gamma > 1$.* Recall@10 and ef =128 are used as representatives throughout. Cache/LLC counters for the varied γ setting are reported in App. F for completeness. Additional setup details are in App. C.

4.1 PERFORMANCE: QUERY EFFICIENCY, CACHE EFFICIENCY, AND RECALL

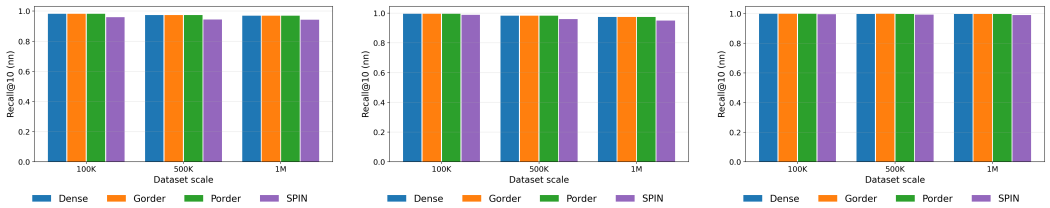
We report the performance of SPIN compared to the baselines by thousand queries executed/sec (kQPS) in Fig. 1, which captures the combined effect of fewer distance computations and reduced cache misses. Table 1 provides the values of raw cache-related metrics, including cache/LLC-references/loads and misses, as well as the number of distance computations (`Ndis`) while executing the test queries.

Query Efficiency. Fig. 1 shows SPIN outperforms dense, Gorder, and Porder across datasets and scales, with gains of roughly $30 - 36\%$ over the dense index, since a sparse neighbor list ensures only the geometrically most relevant neighbors are fetched at runtime. Fewer vectors fetched per node expansion result in fewer cache references, which in turn reduces the overall query time. The graph-reordering methods provide little performance gain over dense, which in turn shows that sparsification combined with optimal ordering ensures higher throughput.

Cache Efficiency. Table 1 shows that for MS MARCO, SPIN experiences fewer cache loads and misses overall compared to the dense or reordered indices. While sparsification reduces memory

Table 1: Cache/LLC and distance/hop counters at dense ef=128 for MS Marco. Sparse indices use $\gamma = 1$. Bold indicates the best (lowest) value per column within each scale. SPIN reduces cache and LLC misses at all scales by requiring fewer distance calculations.

Scale	Method	Cache Refs	Cache Misses	Cache Miss %	LLC Loads	LLC Misses	LLC Miss %	Ndis Mean	Nhops Mean
100K	Dense	3.592e9	5.815e8	16.19%	1.256e9	1.602e8	12.76%	3311.50	**136.20**
	Gorder	3.436e9	5.066e8	14.74%	1.188e9	1.362e8	11.46%	3311.50	**136.20**
	Porder	3.464e9	5.136e8	14.83%	1.204e9	1.374e8	11.41%	3311.50	**136.20**
	SPIN	**2.719e9**	**3.988e8**	**14.67%**	**9.801e8**	**1.057e8**	**10.78%**	**2545.70**	136.60
500K	Dense	4.090e9	8.242e8	20.15%	1.366e9	1.992e8	14.58%	3502.20	**137.70**
	Gorder	3.881e9	7.359e8	18.96%	1.302e9	1.818e8	13.96%	3502.20	**137.70**
	Porder	3.913e9	7.414e8	18.95%	1.319e9	1.788e8	13.55%	3502.20	**137.70**
	SPIN	**3.055e9**	**5.673e8**	**18.57%**	**1.097e9**	**1.368e8**	**12.46%**	**2666.50**	138.50
1M	Dense	4.278e9	9.344e8	21.84%	1.374e9	2.161e8	15.73%	3448.10	**138.40**
	Gorder	4.083e9	8.160e8	**19.99%**	1.304e9	1.763e8	13.53%	3448.10	**138.40**
	Porder	4.116e9	8.348e8	20.28%	1.327e9	1.820e8	13.72%	3448.10	**138.40**
	SPIN	**3.278e9**	**6.584e8**	20.09%	**1.106e9**	**1.389e8**	**12.56%**	**2626.90**	139.20



(a) MS MARCO (b) HotpotQA (c) NQ

Figure 2: Recall@10 at $ef = 128$ across MS MARCO, HotpotQA, and NQ. SPIN produces comparable recall while being much more efficient (please see Sec. 4.1).

references, the optimal memory layout ensures that the geometrically relevant nodes are loaded into the cache together, thereby reducing cache misses at runtime. Experiments on HotpotQA and NQ yield similar results and are presented in App. D.

Recall. Fig. 2 reports Recall@10 at $ef = 128$. SPIN observes a slight drop in recall relative to the baselines ($2 - 4\%$), with the magnitude varying across dataset scales. This is because in some cases the search returns a sub-optimal result, owing to the fact that the increased expected path length in the sparsified graph (as shown in Theorem 3.1) is not compensated by a commensurate increase in the candidate set size (by increasing γ). Thus, in a small number of cases, the search does not explore the optimal results. However, given the 30–36% throughput gain, our experiments indicate that this is a favorable trade-off. Additional results for Recall@1 and Recall@100 are in App. E

4.2 Ablation Study

In the following experiments, we isolate the two main components of SPIN: geometry-aware sparsification and cache-aware reordering. We further examine the throughput-recall trade-off of SPIN under varied γ and empirically validate the theoretical parameters from Sec. 3.

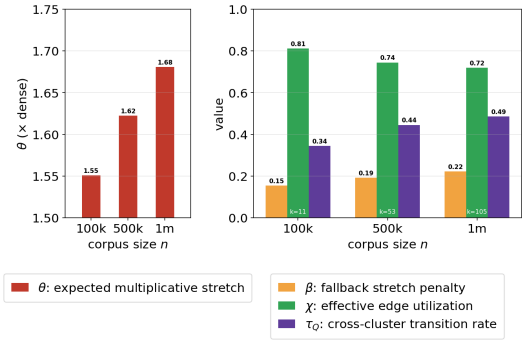
Ablation: Sparsification and Reordering. We isolate SPIN's two components, forming two variants of the sparse index. *Uniform Random Sparsification* deletes edges uniformly rather than being distance-biased, keeping the same budget as SPIN, e.g. $\alpha = 0.3$. *Sparsification without Reordering* skips SPIN's cache-aware reordering, keeping the original memory layout untouched.

Table 2 shows that for MS MARCO, SPIN consistently outperforms both variants across all cache-related measurements. Random sparsification prunes edges regardless of geometric relevance, leading to poor navigability and further $\approx 2\%$ recall drop over SPIN. Alternatively, skipping reordering results in more frequent cache misses and reloads during node expansion, reducing the throughput gains due to sparsification. Experiments on HotpotQA and NQ show similar trends and are presented in App. D.

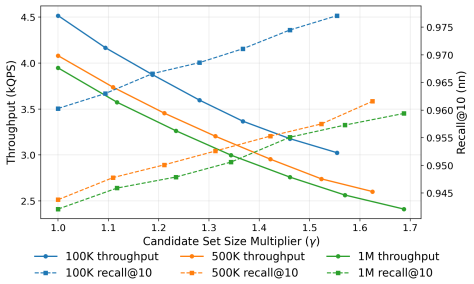
Validating Theoretical Parameters. We empirically measure the parameters of our theoretical characterization (Sec. 2 and App. G) by profiling SPIN's query paths against the dense graph at $\alpha = 0.2$ across graph scales. The key findings in Fig. 3a show that: i) the fallback stretch penalty β, ii) the expected multiplicative stretch under sparsification $\theta := \mathbb{E}\left[\mathcal{S}(G, \widetilde{G})\right]$ (Theorem 3.1), and iii)

Table 2: Cache/LLC and distance/hop counters for MS Marco. Sparse indices use the same ef. Bold indicates the best (lowest) value per column within each scale. SPIN outperforms both variants across most settings, showing that sparsification and reordering each contribute independently.

Scale	Method	Cache Refs	Cache Misses	Cache Miss %	LLC Loads	LLC Misses	LLC Miss %	Ndis Mean	Nhops Mean
100K	SPIN	**2.719e9**	**3.988e8**	**14.67%**	**9.801e8**	**1.057e8**	**10.78%**	**2545.70**	**136.60**
	Random Sparsified	**2.719e9**	4.010e8	14.75%	9.996e8	1.091e8	10.92%	2552.80	**136.60**
	Sparsified Only	2.750e9	4.185e8	15.22%	1.033e9	1.173e8	11.36%	**2545.70**	**136.60**
500K	SPIN	**3.055e9**	**5.673e8**	**18.57%**	**1.097e9**	1.368e8	12.46%	**2666.50**	**138.50**
	Random Sparsified	3.066e9	5.774e8	18.83%	1.108e9	1.447e8	13.05%	2681.50	**138.50**
	Sparsified Only	3.145e9	5.840e8	**18.57%**	1.129e9	**1.367e8**	**12.11%**	**2666.50**	**138.50**
1M	SPIN	**3.278e9**	**6.584e8**	**20.09%**	**1.106e9**	**1.389e8**	**12.56%**	**2626.90**	139.20
	Random Sparsified	3.300e9	6.638e8	20.11%	1.111e9	1.430e8	12.87%	2645.80	**139.10**
	Sparsified Only	3.353e9	6.856e8	20.45%	1.137e9	1.441e8	12.68%	**2626.90**	139.20



(a) Validating analytical parameters at $\alpha = 0.2$.



(b) Throughput vs. Recall@10 for SPIN

Figure 3: (a) The fallback stretch penalty (β), expected multiplicative stretch (θ) and cross-cluster transition rate ($\tau_{\mathcal{Q}}$) increase slightly with scale but remain close to a constant. The effective edge utilization term χ drops with the graph size. This shows that search accuracy and cache performance in SPIN entirely depend on the candidate set size multiplier γ. (b) With increasing values of $\gamma \in [1, \theta]$, recall improves at the cost of lower throughput due to increased cache misses.

the cross-cluster transition rate, $\tau_q = \frac{\mathbb{E}_{q\in\mathcal{Q}}[T_q^{(\gamma)}]}{\mathbb{E}_{q\in\mathcal{Q}}[|P_q^{(\gamma)}|]}$ (Theorem 3.3) increase slightly but can be treated as a constant. We also find the effective edge utilization term, χ (Theorem 3.2) drops with graph size, and can be bounded by a constant. *Therefore, the empirical results support our hypothesis that search accuracy and cache performance on a sparsified, reordered graph produced by* SPIN *can be tuned by a single query-independent parameter, namely the candidate set size multiplier* γ.

Throughput and recall vs. candidate set size multiplier (γ). We test SPIN with varying values of $\gamma \in [1, \theta]$, and measure recall and throughput.Fig. 3b shows that when $\gamma = \theta$, recall for the sparse graph search matches the recall for the dense graph, but throughput drops by about a third relative to $\gamma = 1$. Intermediate values of γ demonstrate the clear tradeoff between throughput and recall. Note that the matching recall at $\gamma = \theta$ implies that $\rho_\theta \approx 0$ and $\delta(\theta)$ is a large value in Theorem 3.2. Further empirical characterization of these parameters are left for future work.

5 CONCLUSION

SPIN is a graph-based nearest-neighbor index that builds on the HNSW architecture and combines geometry-aware sparsification with cache-aware reordering. Across three datasets with up to 1M nodes SPIN improves query throughput by 30–36% over the unmodified baseline and the state of the art cache-aware designs, while sacrificing a small percentage of recall. SPIN's search accuracy and cache performance are both theoretically analyzed and validated in our ablation studies. More broadly, our findings demonstrate that the structure of nearest-neighbor graphs itself provides an important opportunity for improving system efficiency.

AI USE STATEMENT

In this work, we used generative AI tools to assist in the writing of proofs and implement methods. We have not used generative AI tools to help develop theoretical models or conceptual frameworks, formulate mathematical claims, provide critical ingredients for proving mathematical claims, propose or refine hypotheses, design or provide feedback on research methodology or experiments, support qualitative and thematic data analysis, interpret results, and to generate synthetic data sets, assist with translation, clean and reformat datasets is not applicable to this work. Additionally, we used generative AI tools in creation of artifacts, to format references, and identify relevant literature. We have reviewed all AI-assisted work, such as i) all AI-generated codes were reviewed by the authors, and ii) references and literature were cross-checked against conference and journal websites. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

Experimental setup and implementation details are documented throughout the paper, and additional setup details are provided in Appendix C. Code for the sparsification, partitioning, reordering, and evaluation pipeline will be released publicly; for now, it is available at an anonymous repository: `https://anonymous.4open.science/r/SPIN-488A/`.

REFERENCES

A. Akhil and S. G. Zonal HNSW: Scalable approximate nearest neighbor search for billion-scale datasets. In *2025 3rd International Conference on Self Sustainable Artificial Intelligence Systems (ICSSAS)*, pp. 1663–1670, 2025. ISBN null. doi: 10.1109/ICSSAS66150.2025.11081070. ID: 1.

Amazon Web Services. Amazon Bedrock knowledge bases now supports amazon OpenSearch service managed cluster as vector store. `https://aws.amazon.com/blogs/machine-learning/amazon-bedrock-knowledge-bases-now-supports-amazon-opensearch-service-managed-cluster-as-vector-store`, 2025. AWS Machine Learning Blog. Accessed: 2026-09-25.

Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 22–31, 2016. doi: 10.1109/IPDPS.2016.110.

Ilias Azizi, Karima Echihabi, and Themis Palpanas. Graph-based vector search: An experimental evaluation of the state-of-the-art. *Proc. ACM Manag. Data*, 3(1), February 2025. doi: 10.1145/3709693. URL `https://doi.org/10.1145/3709693`.

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. In *Proceedings of the 35th International Conference on Neural Information Processing Systems*, NIPS '21, Red Hook, NY, USA, 2021. Curran Associates Inc. ISBN 9781713845393.

YuAng Chen and Yeh-Ching Chung. Corder: cache-aware reordering for optimizing graph analytics. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, pp. 472–473, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450382946. doi: 10.1145/3437801.3441606. URL `https://doi.org/10.1145/3437801.3441606`.

Benjamin Coleman, Santiago Segarra, Alex Smola, and Anshumali Shrivastava. Graph reordering for cache-efficient near neighbor search. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.

Alex Conway, Laxman Dhulipala, Martin Farach-Colton, Rob Johnson, Ben Landrum, Christopher Musco, Yarin Shechter, Torsten Suel, and Richard Wen. *Efficiently Constructing Sparse Navigable Graphs*, pp. 1631–1665. 2026. doi: 10.1137/1.9781611978971.59. URL `https://epubs.siam.org/doi/abs/10.1137/1.9781611978971.59`.

Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Annual symposium on Computational geometry*, pp. 253–262, 2004.

S. Deng, X. Yan, K. W. N. Kelvin, C. Jiang, and J. Cheng. Pyramid: A general framework for distributed similarity search on large-scale datasets. In *2019 IEEE International Conference on Big Data (Big Data)*, pp. 1066–1071, 2019. ISBN null. doi: 10.1109/BigData47090.2019.9006219. ID: 1.

Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library. *IEEE Transactions on Big Data*, 12(2):346–361, 2026. doi: 10.1109/TBDATA.2025.3618474.

Elastic. Looking back: Elastic's vector search improvements in Elasticsearch & Lucene. https://www.elastic.co/search-labs/blog/vector-search-improvements. Elastic Search Labs blog. Accessed: 2026-09-25.

Wenqi Fan, Yujuan Ding, Liangbo Ning, Shijie Wang, Hengyun Li, Dawei Yin, Tat-Seng Chua, and Qing Li. A survey on rag meeting llms: Towards retrieval-augmented large language models. In *Proceedings of the 30th ACM SIGKDD conference on knowledge discovery and data mining*, pp. 6491–6501, 2024.

M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*, pp. 285–297, 1999. doi: 10.1109/SFFCS.1999.814600.

Hao Guo and Youyou Lu. OdinANN: Direct insert for consistently stable performance in Billion-Scale Graph-Based vector search. In *24th USENIX Conference on File and Storage Technologies (FAST 26)*, pp. 133–147, Santa Clara, CA, February 2026. USENIX Association. ISBN 978-1-939133-53-3. URL https://www.usenix.org/conference/fast26/presentation/guo.

Ben Harwood and Tom Drummond. Fanng: Fast approximate nearest neighbour graphs. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 5713–5722, 2016. doi: 10.1109/CVPR.2016.616.

H. Jin, J. Lee, S. Piao, S. Seo, S. Kwon, and S. Park. Efficient approximate nearest neighbor search via data-adaptive parameter adjustment in hierarchical navigable small graphs. In *2025 Design, Automation & Test in Europe Conference (DATE)*, pp. 1–7, 2025. ISBN 1530-1591. doi: 10.23919/DATE64628.2025.10992903. ID: 1.

D. N. Joanes and C. A. Gill. Comparing measures of sample skewness and kurtosis. *Journal of the Royal Statistical Society: Series D (The Statistician)*, 47(1):183–189, 1998. doi: https://doi.org/10.1111/1467-9884.00122. URL https://rss.onlinelibrary.wiley.com/doi/abs/10.1111/1467-9884.00122.

Jeff Johnson, Matthijs Douze, and Herve Jegou. Billion-Scale Similarity Search with GPUs . *IEEE Transactions on Big Data*, 7(03):535–547, July 2021. ISSN 2332-7790. doi: 10.1109/TBDATA.2019.2921572. URL https://doi.ieeecomputersociety.org/10.1109/TBDATA.2019.2921572.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense passage retrieval for open-domain question answering. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu (eds.), *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 6769–6781, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.550. URL https://aclanthology.org/2020.emnlp-main.550/.

George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998. doi: 10.1137/S1064827595287997. URL https://doi.org/10.1137/S1064827595287997.

Sukjin Kim, Seongyeon Park, Si Ung Noh, Junguk Hong, Taehee Kwon, Hunseong Lim, and Jinho Lee. PathWeaver: A High-Throughput Multi-GPU system for Graph-Based approximate nearest neighbor search. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*, pp. 1501–1517, Boston, MA, July 2025. USENIX Association. ISBN 978-1-939133-48-9. URL https://www.usenix.org/conference/atc25/presentation/kim.

Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, Kristina Toutanova, Llion Jones, Matthew Kelcey, Ming-Wei Chang, Andrew M. Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. Natural questions: A benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:452–466, 2019. doi: 10.1162/tacl_a_00276. URL https://aclanthology.org/Q19-1026/.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc. ISBN 9781713829546.

Binhong Li, Xiao Yan, and Shangqi Lu. Fast-convergent proximity graphs for approximate nearest neighbor search. *Proc. ACM Manag. Data*, 4(1), April 2026. doi: 10.1145/3786650. URL https://doi.org/10.1145/3786650.

Conglong Li, Minjia Zhang, David G. Andersen, and Yuxiong He. Improving approximate nearest neighbor search through learned adaptive early termination. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pp. 2539–2554, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367356. doi: 10.1145/3318464.3380600. URL https://doi.org/10.1145/3318464.3380600.

Zhonggen Li, Xiangyu Ke, Yifan Zhu, Bocheng Yu, Baihua Zheng, and Yunjun Gao. Scalable graph indexing using gpus for approximate nearest neighbor search. *Proc. ACM Manag. Data*, 3(6), December 2025. doi: 10.1145/3769825. URL https://doi.org/10.1145/3769825.

Yu A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.

Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45:61–68, 2014. ID: 271528.

Magdalen Dobson Manohar, Zheqi Shen, Guy Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. Parlayann: Scalable and deterministic parallel graph-based approximate nearest neighbor search algorithms. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '24, pp. 270–285, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704352. doi: 10.1145/3627535.3638475. URL https://doi.org/10.1145/3627535.3638475.

Meta AI Research. Faiss indexes. https://github.com/facebookresearch/faiss/wiki/Faiss-indexes. Faiss wiki. Accessed: 2026-09-25.

Tai Ming, Xin Hu, and Yimin Wu. Liteqg: Towards scalable and memory-efficient graph-based approximate nearest neighbor search. In De-Shuang Huang, Yijie Pan, Wei Chen, and Bo Li (eds.), *Advanced Intelligent Computing Technology and Applications*, pp. 3–18, Singapore, 2025. Springer Nature Singapore. ISBN 978-981-96-9901-8.

MongoDB. Atlas Vector Search. https://www.mongodb.com/products/platform/atlas-vector-search. Accessed: 2026-09-25.

P. Muppuri. Optimized framework for merging navigable small-world graphs: Balancing efficiency and search performance. In *2026 IEEE International Conference on Interdisciplinary Approaches in Technology and Management for Social Innovation (IATMSI)*, volume 4th, pp. 1–6, 2026. ISBN null. doi: 10.1109/IATMSI68868.2026.11465357. ID: 1.

Tri Nguyen, Mir Rosenberg, Xia Song, Jianfeng Gao, Saurabh Tiwary, Rangan Majumder, and Li Deng. Ms marco: A human generated machine reading comprehension dataset. November 2016. URL https://www.microsoft.com/en-us/research/publication/ms-marco-human-generated-machine-reading-comprehension-dataset/.

Alexander Ponomarenko. Query-based improvement procedure and self-adaptive graph construction algorithm for approximate nearest neighbor search, 2015.

Miloš Radovanović. Hubs in nearest-neighbor graphs: Origins, applications and challenges. In *Proceedings of the 8th International Conference on Web Intelligence, Mining and Semantics*, WIMS '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450354899. doi: 10.1145/3227609.3227691. URL https://doi.org/10.1145/3227609.3227691.

Miloš Radovanović, Alexandros Nanopoulos, and Mirjana Ivanović. Nearest neighbors in high-dimensional data: the emergence and influence of hubs. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pp. 865–872, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605585161. doi: 10.1145/1553374.1553485. URL https://doi.org/10.1145/1553374.1553485.

Miloš Radovanović, Alexandros Nanopoulos, and Mirjana Ivanović. Hubs in space: Popular nearest neighbors in high-dimensional data. *J. Mach. Learn. Res.*, 11:2487–2531, December 2010. ISSN 1532-4435.

Redis. Vector search concepts. https://redis.io/docs/latest/develop/ai/search-and-query/vectors/. Redis documentation. Accessed: 2026-09-25.

Jiancheng Ruan, Tingyang Chen, Renchi Yang, Xiangyu Ke, and Yunjun Gao. Empowering graph-based approximate nearest neighbor search with adaptive awareness capabilities. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*, KDD '25, pp. 2444–2454, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400714542. doi: 10.1145/3711896.3736930. URL https://doi.org/10.1145/3711896.3736930.

Ziwen Song, Bin Wang, and Xiaochun Yang. Accelerating high-dimensional ann search via skipping redundant distance computations. *Proc. ACM Manag. Data*, 3(6), December 2025. doi: 10.1145/3769753. URL https://doi.org/10.1145/3769753.

C. Spearman. The proof and measurement of association between two things. *The American Journal of Psychology*, 15(1):72–101, 1904. ISSN 00029556. URL http://www.jstor.org/stable/1412159.

Suhas Jayaram Subramanya, Devvrit, Rohan Kadekodi, Ravishankar Krishaswamy, and Harsha Vardhan Simhadri. *DiskANN: fast accurate billion-point nearest neighbor search on a single node*. Curran Associates Inc., Red Hook, NY, USA, 2019.

Tommaso Teofili and Jimmy Lin. Patience in proximity: A simple early termination strategy for HNSW graph traversal in approximate k-nearest neighbor search, 2025.

Nenad Tomašev, Miloš Radovanović, Dunja Mladenić, and Mirjana Ivanović. *Hubness-Based Clustering of High-Dimensional Data*, pp. 353–386. Springer International Publishing, Cham, 2015. ISBN 978-3-319-09259-1. doi: 10.1007/978-3-319-09259-1_11. URL https://doi.org/10.1007/978-3-319-09259-1_11.

Karthik Venkatasubba, Saim Khan, Somesh Singh, Harsha Vardhan Simhadri, and Jyothi Vedurada. Bang: Billion-scale approximate nearest neighbour search using a single gpu. *IEEE Transactions on Big Data*, 11(6):3142–3157, 2025. doi: 10.1109/TBDATA.2025.3581085.

Gang Wang, Xingxing Ke, Jian Hu, Qianling Li, Minghao Shao, and Jianfeng Fan. Learning to prune: General and efficient approximate nearest neighbor search with direction navigating graph. In *Proceedings of the 2022 5th International Conference on Data Storage and Data Engineering*, DSDE '22, pp. 50–58, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450395724. doi: 10.1145/3528114.3528123. URL https://doi.org/10.1145/3528114.3528123.

Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *Proc. VLDB Endow.*, 14(11):1964–1978, 2021. ISSN 2150-8097. doi: 10.14778/3476249.3476255. URL https://doi.org/10.14778/3476249.3476255.

Mengzhao Wang, Lingwei Lv, Xiaoliang Xu, Yuxiang Wang, Qiang Yue, and Jiongkang Ni. An efficient and robust framework for approximate nearest neighbor search with attribute constraint. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 15738–15751. Curran Associates, Inc., 2023. doi: 10.52202/075280-0692. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/32e41d6b0a51a63a9a90697da19d235d-Paper-Conference.pdf.

Mengzhao Wang, Weizhi Xu, Xiaomeng Yi, Songlin Wu, Zhangyang Peng, Xiangyu Ke, Yunjun Gao, Xiaoliang Xu, Rentong Guo, and Charles Xie. Starling: An i/o-efficient disk-resident graph index framework for high-dimensional vector similarity search on data segment. *Proc. ACM Manag. Data*, 2(1), March 2024. doi: 10.1145/3639269. URL https://doi.org/10.1145/3639269.

Mengzhao Wang, Haotian Wu, Xiangyu Ke, Yunjun Gao, Yifan Zhu, and Wenchao Zhou. Accelerating graph indexing for anns on modern cpus. *Proc. ACM Manag. Data*, 3(3), June 2025. doi: 10.1145/3725260. URL https://doi.org/10.1145/3725260.

Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pp. 1813–1828, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450335317. doi: 10.1145/2882903.2915220. URL https://doi.org/10.1145/2882903.2915220.

Jiadong Xie, Jeffrey Xu Yu, and Yingfan Liu. Graph based k-nearest neighbor search revisited. *ACM Trans. Database Syst.*, 50(4), June 2025. ISSN 0362-5915. doi: 10.1145/3736716. URL https://doi.org/10.1145/3736716.

Xiaoliang Xu, Haonan Dai, Can Li, Mengzhao Wang, and Qiang Yue. Through the lens of hubness: A revisit on graph-based approximate nearest neighbor search: [experiments & analysis]. *Proc. ACM Manag. Data*, 4(3), May 2026. doi: 10.1145/3802120. URL https://doi.org/10.1145/3802120.

Chengzhi Yan and Hongya Wang. Hierarchical clustering-based high-performance adjacency graph search algorithm, 2025.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (eds.), *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2369–2380, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1259. URL https://aclanthology.org/D18-1259/.

Hanchao Zheng, Yang Chen, and Zhe Wu. Label-balanced graph index for filtered approximate nearest neighbor search with low-frequency labels. In *Proceedings of the 38th International Conference on Scalable Scientific Data Management*, SSDBM '26, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400727085. doi: 10.1145/3828820.3828830. URL https://doi.org/10.1145/3828820.3828830.

Zulun Zhu, Tiancheng Huang, Kai Wang, Junda Ye, Xinghe Chen, and Siqiang Luo. Graph-based approaches and functionalities in retrieval-augmented generation: A comprehensive survey. *ACM Comput. Surv.*, 58(10), April 2026. ISSN 0360-0300. doi: 10.1145/3795880. URL https://doi.org/10.1145/3795880.

A HNSW STRUCTURAL DIAGNOSTICS

In this section, we report the measurements of the two structural properties of HNSW built on the MS MARCO dataset that motivate the design Sec. 2 of SPIN: `skewed hubness` and `uneven edge importance`. The in-degree distribution of the layer-0 graph is right-skewed at every scale as can be seen in table 3. The skewness increases with scale as a small fraction of hub nodes holds a large proportion of in-edges. For the edge importance experiment, we bin the layer-0 edges by geometric length and present the result of traversal in table 4. A profiling experiment with 5000 queries shows that the longest 30% of edges are traversed only $0.4\times$ the shortest 30% of edges. Moreover, a negative rank correlation prevails between edge length and traversal frequency across all scales. Since long-range edges contribute less to navigability during traversal, they are penalized heavily in SPIN.

Table 3: In-degree distribution of the level-0 HNSW graph across scales. *skew* is Fisher's moment coefficient (Joanes & Gill, 1998) of skewness (the third standardized moment) of the in-degree distribution; a positive value indicates a right tail of a few *hub* nodes, and larger values mean stronger hubness. The number of hub nodes grows with scale.

Dataset	Scale	N	mean deg	max in-deg	skew	top-1% (hubs)
msmarco	100k	100,000	32.4	285	1.76	0.037
msmarco	500k	500,000	31.0	485	2.37	0.044
msmarco	1m	1,000,000	29.8	632	2.67	0.048

Table 4: Layer-0 edges grouped by geometric length (L_2 between endpoint embeddings) into the shortest 30%, middle 40%, and longest 30% of edges. For each group, we report its traversal frequency relative to the short group (short $= 1.00$) and its share of total traversal mass. ρ is Spearman's rank correlation (Spearman, 1904) edge length and per-edge traversal frequency over all level-0 edges ($\rho \in [-1, 1]$; negative means longer edges are traversed less; all $p \approx 0$). Long-range edges are traversed far less than short-range edges at every scale, confirming they contribute little once the search reaches the correct region.

		rel. traversal freq			mass share (%)			
Dataset	Scale	short	mid	long	short	mid	long	ρ
msmarco	100k	1.00	0.64	0.43	43.9	37.4	18.7	-0.30
msmarco	500k	1.00	0.59	0.36	45.9	36.7	17.5	-0.22
msmarco	1m	1.00	0.61	0.44	46.8	36.3	17.0	-0.18

B ADDITIONAL RELATED WORK

A parallel algorithm-focused line of work studies how to optimize traversal costs by reducing node expansion and leveraging informed edge selection while processing queries. These strategies include: skipping redundant distance checks (Song et al., 2025), learning dynamic pruning rules (Wang et al., 2022), applying early-stopping criteria (Li et al., 2020; Teofili & Lin, 2025), or adapting parameter settings on the fly (Jin et al., 2025). In proximity graphs, these unnecessary traversals are often caused by the hierarchical nature of the graphs or by the emergence of *hub* nodes (Radovanović et al., 2009; 2010; Tomašev et al., 2015; Radovanović, 2018; Xu et al., 2026) that reside on the common pathways between almost every pair of nodes.

C ADDITIONAL DETAILS ON EXPERIMENTAL SETUP

Hardware and Measurement Protocol. The setup steps: index construction, graph sparsification, partitioning, reordering, and other cache-independent experiments were run on CPU-only nodes of the NCSA Delta cluster. All cache and timing measurements are conducted on a single workstation

with a 30MB shared L3 cache and 32GB of RAM, running Linux OS. The search thread is pinned to a single performance core, and the page cache is warmed before every timed run. For each operating point (arm, ef), 5 iterations are run and the first one is discarded. Each iteration uses 5 warm-up and 10 timed repetitions over a fixed 1,000-query subsample across datasets and scales. Hardware counters are read via Linux `perf` in a *single* event group: `cache-references`, `cache-misses`, `L1-dcache-loads`, `L1-dcache-load-misses`, `LLC-loads`, `LLC-load-misses`, and `dTLB-load-misses`.

Dense arms are swept over $ef \in \{24, 32, 48, 64, 96, 128, 192, 256, 400\}$, and the sparse ef is stretched following the sparsification rate $\alpha = 0.30$. We report metrics at matched ef and varied γ values.

D ADDITIONAL CACHE RESULTS UNDER MATCHED ef

Here, we repeat the matched-ef cache/LLC evaluation of Sec. 4.1 on the two remaining datasets in our benchmark, HotpotQA and NQ. These evaluations aim to confirm that the trends in cache efficiency generalize beyond MS MARCO.

Cache/LLC Counters Tables 5 and 6 show SPIN reduces cache references, cache misses, LLC loads, and LLC misses relative to dense, Gorder, and Porder at every scale, with fewer distance calculations in query time. This follows the pattern observed in MS MARCO. Since HotpotQA and NQ's passages and queries are embedded and evaluated under the same pipeline, the results provide evidence that geometry-aware sparsification can be adopted as a general method for efficient query processing across datasets.

Table 5: Cache/LLC and distance/hop counters at dense ef=128 for HotpotQA. Sparse indices use the same ef. Bold indicates the best (lowest) value per column within each scale. SPIN reduces cache and LLC misses at all scales, confirming the gain is not dataset-dependent.

Scale	Method	Cache Refs	Cache Misses	Cache Miss %	LLC Loads	LLC Misses	LLC Miss %	Ndis Mean	Nhops Mean
100K	Dense	3.582e9	5.704e8	15.93%	1.261e9	1.517e8	12.03%	3316.70	**136.00**
	Gorder	3.435e9	4.956e8	14.43%	1.195e9	1.298e8	10.86%	3316.70	**136.00**
	Porder	3.406e9	4.888e8	**14.35%**	1.180e9	1.263e8	10.70%	3316.70	**136.00**
	SPIN	**2.731e9**	**4.002e8**	14.66%	**9.885e8**	**1.040e8**	**10.52%**	**2569.90**	136.20
500K	Dense	4.454e9	9.203e8	20.66%	1.443e9	2.225e8	15.42%	3830.40	**137.30**
	Gorder	4.228e9	7.926e8	18.75%	1.353e9	1.873e8	13.85%	3830.40	**137.30**
	Porder	4.186e9	7.817e8	18.67%	1.339e9	1.832e8	13.68%	3830.40	**137.30**
	SPIN	**3.320e9**	**6.184e8**	**18.62%**	**1.175e9**	**1.445e8**	**12.30%**	**2925.40**	137.80
1M	Dense	4.983e9	1.141e9	22.89%	1.491e9	2.517e8	16.88%	4044.70	**137.90**
	Gorder	4.729e9	1.015e9	21.46%	1.412e9	2.446e8	17.33%	4044.70	**137.90**
	Porder	4.688e9	9.798e8	20.90%	1.390e9	2.238e8	16.10%	4044.70	**137.90**
	SPIN	**3.705e9**	**7.601e8**	**20.52%**	**1.243e9**	**1.647e8**	**13.26%**	**3054.90**	138.50

Table 6: Cache/LLC and distance/hop counters at dense ef=128 for NQ. Sparse indices use the same ef. Bold indicates the best (lowest) value per column within each scale. NQ shows the same cache/LLC improvement as MS MARCO and HotpotQA.

Scale	Method	Cache Refs	Cache Misses	Cache Miss %	LLC Loads	LLC Misses	LLC Miss %	Ndis Mean	Nhops Mean
100K	Dense	2.969e9	4.423e8	14.90%	1.089e9	1.186e8	10.89%	2867.30	**135.90**
	Gorder	2.956e9	4.395e8	14.87%	1.077e9	1.166e8	10.83%	2867.30	**135.90**
	Porder	2.959e9	4.382e8	**14.81%**	1.076e9	1.144e8	**10.63%**	2867.30	**135.90**
	SPIN	**2.254e9**	**3.409e8**	15.12%	**8.100e8**	**9.219e7**	11.38%	**2188.20**	136.20
500K	Dense	3.545e9	6.966e8	19.65%	1.242e9	1.806e8	14.54%	3185.80	**137.20**
	Gorder	3.519e9	6.702e8	19.04%	1.205e9	1.649e8	13.68%	3185.80	**137.20**
	Porder	3.508e9	6.591e8	**18.79%**	1.213e9	1.593e8	13.14%	3185.80	**137.20**
	SPIN	**2.713e9**	**5.105e8**	18.82%	**9.599e8**	**1.300e8**	**13.54%**	**2418.30**	137.60
1M	Dense	3.980e9	8.246e8	20.72%	1.304e9	1.841e8	14.11%	3328.00	**137.30**
	Gorder	3.932e9	7.998e8	20.34%	1.276e9	1.786e8	14.00%	3328.00	**137.30**
	Porder	3.928e9	8.081e8	20.57%	1.277e9	1.790e8	14.02%	3328.00	**137.30**
	SPIN	**3.112e9**	**6.271e8**	**20.15%**	**1.035e9**	**1.324e8**	**12.79%**	**2535.60**	137.70

Sparsification and Reordering Ablation We also repeat the sparsification/reordering ablation (Sec. 4.2) on HotpotQA and NQ to verify if the components show similar contributions towards

performance gains on these datasets as well. The results in Tables 7 and 8 exhibit mostly similar patterns as MS MARCO. However, the margins are narrower on NQ than at other datasets. NQ's passages are shorter and more heterogeneous than MS MARCO, which likely produces a less skewed distribution of edge similarity that the geometry-aware sparsification process cannot fully exploit.

Table 7: Cache/LLC and distance/hop counters at dense ef=128 for HotpotQA. Sparse indices use the same ef. Bold indicates the best (lowest) value per column within each scale. The performance trend matches that of MS MARCO, with narrower margins.

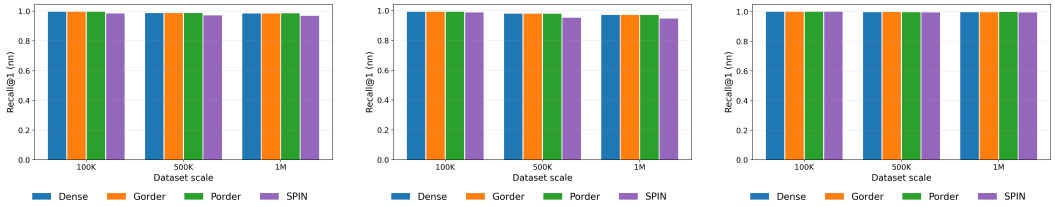
Scale	Method	Cache Refs	Cache Misses	Cache Miss %	LLC Loads	LLC Misses	LLC Miss %	Ndis Mean	Nhops Mean
100K	SPIN	**2.731e9**	**4.002e8**	14.66%	**9.885e8**	1.040e8	10.52%	**2569.90**	**136.20**
	Random Sparsified	2.743e9	4.014e8	**14.63%**	9.949e8	**1.037e8**	**10.42%**	2578.90	**136.20**
	Sparsified Only	2.791e9	4.157e8	14.90%	1.020e9	1.085e8	10.64%	**2569.90**	**136.20**
500K	SPIN	**3.320e9**	**6.184e8**	18.62%	**1.175e9**	**1.445e8**	12.30%	**2925.40**	**137.80**
	Random Sparsified	3.345e9	6.220e8	**18.60%**	1.180e9	1.452e8	12.31%	2944.50	**137.80**
	Sparsified Only	3.424e9	6.480e8	18.92%	1.224e9	1.484e8	**12.12%**	**2925.40**	**137.80**
1M	SPIN	**3.705e9**	**7.601e8**	**20.52%**	**1.243e9**	**1.647e8**	**13.26%**	**3054.90**	**138.50**
	Random Sparsified	3.751e9	7.760e8	20.69%	1.250e9	1.740e8	13.92%	3093.90	**138.50**
	Sparsified Only	3.827e9	8.064e8	21.07%	1.295e9	1.723e8	13.31%	**3054.90**	**138.50**

Table 8: Cache/LLC and distance/hop counters at dense ef=128 for NQ. Sparse indices use the same ef. Bold indicates the best (lowest) value per column within each scale. Both components help NQ, with narrower margins.

Scale	Method	Cache Refs	Cache Misses	Cache Miss %	LLC Loads	LLC Misses	LLC Miss %	Ndis Mean	Nhops Mean
100K	SPIN	**2.254e9**	3.409e8	15.12%	8.100e8	9.219e7	11.38%	**2188.20**	**136.20**
	Random Sparsified	2.315e9	3.436e8	14.84%	8.236e8	9.081e7	11.03%	2235.40	**136.20**
	Sparsified Only	2.280e9	**3.280e8**	**14.39%**	**8.025e8**	**8.468e7**	**10.55%**	**2188.20**	**136.20**
500K	SPIN	**2.713e9**	**5.105e8**	18.82%	9.599e8	1.300e8	13.54%	**2418.30**	**137.60**
	Random Sparsified	2.772e9	5.163e8	**18.63%**	9.808e8	1.267e8	12.92%	2463.60	**137.60**
	Sparsified Only	2.732e9	5.107e8	18.69%	**9.575e8**	**1.215e8**	**12.69%**	**2418.30**	**137.60**
1M	SPIN	**3.112e9**	**6.271e8**	**20.15%**	1.035e9	**1.324e8**	**12.79%**	**2535.60**	**137.70**
	Random Sparsified	3.144e9	6.362e8	20.24%	1.061e9	1.368e8	12.90%	2568.90	**137.70**
	Sparsified Only	3.124e9	6.353e8	20.34%	**1.034e9**	1.363e8	13.19%	**2535.60**	**137.70**

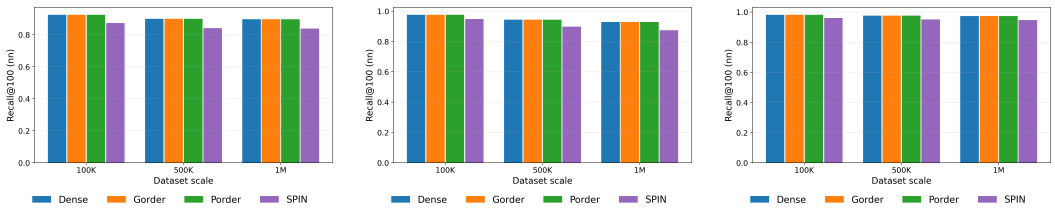
E ADDITIONAL RECALL RESULTS

Here, we report Recall@1 and Recall@100 at $ef = 128$, across all datasets and scales, to verify that the same trends hold as Recall@10, reported in the main text (Sec. 4.1) when the cutoffs are increased or decreased. Figures 4 and 5 show that SPIN experiences a drop in both recall@1 and recall@100 following the same pattern as recall@10. The gap is somewhat larger at Recall@100 than the other two measurements. Since SPIN's geometry-aware pruning is designed to retain edges to the closest neighbors, the lower-ranked true neighbors living farther and in diverse regions of the graph sometimes become unreachable if searched under a restrained ef. As the closest neighbors remain generally reachable, the top-1 results are hardly affected. This gap can be closed if the search is conducted under a more relaxed setting with $\gamma > 1$ as demonstrated in 4.2.



(a) MS MARCO (b) HotpotQA (c) NQ

Figure 4: Recall@1 at $ef = 128$ across MS MARCO, HotpotQA, and NQ. The recall gap is smallest here compared to Recall@10 and Recall@100.



(a) MS MARCO (b) HotpotQA (c) NQ

Figure 5: Recall@100 at $ef = 128$ across MS MARCO, HotpotQA, and NQ.Shows the largest recall gap, as SPIN penalizes lower-ranked true neighbors heavily.

F CACHE EFFICIENCY UNDER VARIED γ

In Sec. 4.2, we vary γ to see if SPIN's recall gap can be closed by searching under a broader setting with a higher ef. Here, we report the underlying cache/LLC counters at various γ's after the dense-matched ef up to the theoretical upper bound, θ, derived (4.2) across various scales in the MS MARCO dataset empirically. We also repeat the sparsification/reordering ablation at each γ. Tables 9 and 10 show the results for these experiments. Since the most important component of SPIN's cache advantage comes from sparsification, SPIN steadily loses its advantage over the dense indices as it is forced to search a broader region of the graph under a higher ef. In this setting, SPIN constantly has to fetch, retain, and search through a larger set of candidate nodes compared to the dense indices, which increases its memory references, and, since the retained nodes belong to a broader graph region, the references incur more cache misses that cannot be recovered by reordering alone. Thus, the throughput gain degrades rapidly, well before reaching the theoretical upper bound. However, the results of ablation studies remain consistent as they are conducted along the sparse axis.

G PROOFS

G.1 DEFINING SEARCH PATH LENGTH INCREASE

Proof Outline: The first step in our algorithm is a probabilistic sparsification of the graph, under constraints of connectedness. Intuitively, sparsification can result in important edges being removed that otherwise lead to shorter paths between a pair of nodes in the dense graph. Therefore, the first quantity we measure is the *multiplicative stretch* in the sparse graph – the expected increase (as a ratio against the dense graph) in the path length between two nodes in the graph that were otherwise connected with a direct edge. A large multiplicative stretch implies that during a search, more nodes must be visited to maintain similar search accuracy. We show that under measurable quantities for a given NN graph, the multiplicative stretch is bounded by a constant.

To compensate for the sparsification, after deriving a fixed multiplicative stretch, we define a multiplicative parameter γ to expand the size of the candidate set (ef) maintained during the search on the sparse graph. With this increased candidate set size, we show that the search on the sparsified graph in SPIN returns either the same result or a better candidate (closer to the query point) as the search on the dense graph, with the probability scaling as a function in γ.

Reachability assumption: Sparsification can affect reachability between two nodes in the graph. To simplify our proofs, we make an assumption about the reachable set in the sparse graph compared to the dense graph. Specifically, we define the event

$$\mathcal{R} = \left\{ \frac{\sum_{v \in V} |\mathcal{R}_{\widetilde{G}}(v)|}{\sum_{v \in V} |\mathcal{R}_G|} \geq 1 - \epsilon \right\}$$

, where ϵ is upper-bounded by a small constant ϵ_0. Our empirical analysis shows that the value of ϵ_0 remained bounded by a value $\leq 10^{-3}$ for graphs containing up to 1 million nodes. **Thus, all our**

Table 9: Cache/LLC and distance/hop counters at dense ef=128 for MS Marco across different values of the candidate set size multiplier (γ). Bold indicates the best (lowest) value per column within each scale. SPIN's recall improves with increasing the candidate set size.

Scale	Method	Cache Refs	Cache Misses	Cache Miss %	LLC Loads	LLC Misses	LLC Miss %	Ndis Mean	Nhops Mean
$\gamma = \{1.0918, 1.1037, 1.1135\}$ @ $\{100k, 500k, 1M\}$									
100K	Dense	3.587e9	5.759e8	**16.05%**	1.268e9	1.631e8	12.86%	3311.50	**136.20**
	Gorder	3.436e9	4.963e8	14.44%	1.194e9	1.331e8	11.15%	3311.50	**136.20**
	Porder	3.468e9	5.108e8	14.73%	1.201e9	1.349e8	11.23%	3311.50	**136.20**
	SPIN	**2.932e9**	**4.296e8**	14.65%	**1.041e9**	**1.128e8**	**10.83%**	**2735.50**	148.50
500K	Dense	4.101e9	7.910e8	19.29%	1.376e9	1.772e8	12.88%	3502.20	**137.70**
	Gorder	3.887e9	6.984e8	**17.97%**	1.315e9	1.600e8	12.17%	3502.20	**137.70**
	Porder	3.929e9	7.196e8	18.31%	1.317e9	1.643e8	12.47%	3502.20	**137.70**
	SPIN	**3.337e9**	**6.032e8**	18.08%	**1.170e9**	**1.366e8**	**11.68%**	**2902.60**	152.30
1M	Dense	4.292e9	9.118e8	21.24%	1.390e9	1.938e8	13.94%	3448.10	**138.40**
	Gorder	4.084e9	8.111e8	**19.86%**	1.310e9	1.742e8	13.30%	3448.10	**138.40**
	Porder	4.119e9	8.303e8	20.16%	1.324e9	1.785e8	13.48%	3448.10	**138.40**
	SPIN	**3.568e9**	**7.158e8**	20.06%	**1.186e9**	**1.490e8**	**12.57%**	**2879.80**	154.00
$\gamma = \{1.1836, 1.2075, 1.2269\}$ @ $\{100k, 500k, 1M\}$									
100K	Dense	3.599e9	5.659e8	15.72%	1.259e9	1.526e8	12.12%	3311.50	**136.20**
	Gorder	3.440e9	4.952e8	**14.40%**	1.191e9	1.314e8	**11.03%**	3311.50	**136.20**
	Porder	3.470e9	5.095e8	14.68%	1.201e9	1.342e8	11.18%	3311.50	**136.20**
	SPIN	**3.136e9**	**4.669e8**	14.89%	**1.098e9**	**1.229e8**	11.19%	**2922.60**	160.40
500K	Dense	4.101e9	7.923e8	19.32%	1.376e9	1.777e8	12.91%	3502.20	**137.70**
	Gorder	3.887e9	6.986e8	**17.97%**	1.314e9	1.600e8	12.18%	3502.20	**137.70**
	Porder	3.924e9	7.181e8	18.30%	1.329e9	1.646e8	12.39%	3502.20	**137.70**
	SPIN	**3.584e9**	**6.564e8**	18.32%	**1.235e9**	**1.499e8**	**12.13%**	**3120.80**	165.20
1M	Dense	4.296e9	9.104e8	21.19%	1.385e9	1.916e8	13.83%	3448.10	**138.40**
	Gorder	4.082e9	8.124e8	**19.90%**	1.317e9	1.746e8	**13.26%**	3448.10	**138.40**
	Porder	4.117e9	8.308e8	20.18%	1.331e9	1.785e8	13.41%	3448.10	**138.40**
	SPIN	**3.853e9**	**7.822e8**	20.30%	**1.264e9**	**1.681e8**	13.30%	**3127.60**	168.70
$\gamma = \{1.2754, 1.3112, 1.3404\}$ @ $\{100k, 500k, 1M\}$									
100K	Dense	3.602e9	5.635e8	15.65%	1.262e9	1.500e8	11.89%	3311.50	**136.20**
	Gorder	3.439e9	**4.949e8**	**14.39%**	1.197e9	1.313e8	**10.97%**	3311.50	**136.20**
	Porder	3.469e9	5.108e8	14.72%	1.202e9	1.348e8	11.22%	3311.50	**136.20**
	SPIN	**3.347e9**	5.055e8	15.10%	**1.146e9**	**1.311e8**	11.44%	**3106.90**	172.40
500K	Dense	4.102e9	7.909e8	19.28%	1.376e9	1.773e8	12.88%	3502.20	**137.70**
	Gorder	3.888e9	**6.994e8**	**17.99%**	1.314e9	**1.615e8**	**12.29%**	3502.20	**137.70**
	Porder	3.923e9	7.204e8	18.36%	1.330e9	1.668e8	12.54%	3502.20	**137.70**
	SPIN	**3.829e9**	7.120e8	18.59%	**1.292e9**	1.640e8	12.70%	**3336.20**	178.00
1M	Dense	4.295e9	9.111e8	21.21%	1.378e9	1.913e8	13.88%	3448.10	**138.40**
	Gorder	**4.088e9**	**8.136e8**	**19.90%**	1.305e9	1.765e8	13.53%	3448.10	**138.40**
	Porder	4.120e9	8.286e8	20.11%	1.317e9	**1.754e8**	**13.31%**	3448.10	**138.40**
	SPIN	4.119e9	8.462e8	20.55%	**1.316e9**	1.809e8	13.74%	**3358.00**	182.70
$\gamma = \{1.3671, 1.4149, 1.4538\}$ @ $\{100k, 500k, 1M\}$									
100K	Dense	3.602e9	5.630e8	15.63%	1.258e9	1.496e8	11.90%	3311.50	**136.20**
	Gorder	**3.442e9**	**4.958e8**	**14.41%**	**1.190e9**	**1.310e8**	**11.01%**	3311.50	**136.20**
	Porder	3.468e9	5.081e8	14.65%	1.206e9	1.348e8	11.18%	3311.50	**136.20**
	SPIN	3.519e9	5.461e8	15.52%	1.195e9	1.463e8	12.24%	**3274.10**	183.30
500K	Dense	**4.102e9**	7.909e8	19.28%	**1.376e9**	1.772e8	12.87%	**3502.20**	**137.70**
	Gorder	3.887e9	**6.994e8**	**18.00%**	1.315e9	**1.608e8**	**12.23%**	**3502.20**	**137.70**
	Porder	3.925e9	7.196e8	18.34%	1.330e9	1.652e8	12.42%	**3502.20**	**137.70**
	SPIN	4.098e9	7.779e8	18.98%	1.344e9	1.804e8	13.43%	3568.60	192.00
1M	Dense	**4.291e9**	9.076e8	21.15%	**1.380e9**	1.884e8	13.65%	**3448.10**	**138.40**
	Gorder	4.085e9	**8.096e8**	**19.82%**	1.301e9	**1.727e8**	**13.27%**	**3448.10**	**138.40**
	Porder	4.128e9	8.356e8	20.24%	1.325e9	1.759e8	13.28%	**3448.10**	**138.40**
	SPIN	4.406e9	9.281e8	21.06%	1.376e9	2.021e8	14.68%	3607.40	197.80
$\gamma = \{1.5507, 1.6224, 1.6807\}$ @ $\{100k, 500k, 1M\}$									
100K	Dense	6.885e9	5.855e8	8.51%	2.556e9	1.4195e8	5.55%	**3316.80**	**136.10**
	Gorder	**6.580e9**	**5.025e8**	**7.63%**	**2.4205e9**	**1.2365e8**	**5.11%**	**3316.80**	**136.10**
	Porder	6.635e9	5.205e8	7.84%	2.4515e9	1.268e8	5.17%	**3316.80**	**136.10**
	SPIN	7.520e9	6.660e8	8.86%	2.532e9	1.6045e8	6.34%	3630.40	207.10
500K	Dense	7.705e9	8.375e8	10.87%	2.8065e9	1.833e8	6.53%	**3509.70**	**137.70**
	Gorder	**7.320e9**	**7.200e8**	**9.84%**	**2.6005e9**	**1.6115e8**	**6.20%**	**3509.70**	**137.70**
	Porder	7.395e9	7.445e8	10.06%	2.6225e9	1.645e8	6.27%	**3509.70**	**137.70**
	SPIN	8.670e9	1.0055e9	11.60%	2.7615e9	2.2615e8	8.19%	3989.40	217.70
1M	Dense	7.965e9	9.400e8	11.81%	2.7325e9	1.935e8	7.08%	**3461.90**	**138.20**
	Gorder	**7.565e9**	**8.025e8**	**10.60%**	**2.5755e9**	**1.7205e8**	**6.68%**	**3461.90**	**138.20**
	Porder	7.645e9	8.330e8	10.89%	2.5965e9	1.7705e8	6.82%	**3461.90**	**138.20**
	SPIN	9.245e9	1.1965e9	12.94%	2.834e9	2.616e8	9.23%	4073.80	226.60

Table 10: Cache/LLC and distance/hop counters at dense ef=128 for MS Marco across different values of the candidate set size multiplier (γ). Bold indicates the best (lowest) value per column within each scale. (ablation: geometry-aware vs. random sparsification vs. no reordering). SPIN mostly outperforms the other variants as they reside in the same sparse regions.

Scale	Method	Cache Refs	Cache Misses	Cache Miss %	LLC Loads	LLC Misses	LLC Miss %	Ndis Mean	Nhops Mean
$\gamma = \{1.0918, 1.1037, 1.1135\}$ @ $\{100k, 500k, 1M\}$									
100K	SPIN	2.932e9	**4.296e8**	**14.65%**	**1.041e9**	1.128e8	**10.83%**	**2735.50**	**148.50**
	Random Sparsified	**2.925e9**	4.349e8	14.87%	1.070e9	1.216e8	11.37%	2744.10	**148.50**
	Sparsified Only	2.976e9	4.442e8	14.92%	1.090e9	**1.212e8**	11.12%	**2735.50**	**148.50**
500K	SPIN	**3.337e9**	**6.032e8**	**18.08%**	**1.170e9**	**1.366e8**	11.68%	**2902.60**	**152.30**
	Random Sparsified	3.360e9	6.104e8	18.17%	1.177e9	1.392e8	11.82%	2920.00	152.40
	Sparsified Only	3.428e9	6.261e8	18.27%	1.210e9	1.374e8	**11.35%**	**2902.60**	**152.30**
1M	SPIN	**3.568e9**	**7.158e8**	**20.06%**	**1.186e9**	**1.490e8**	**12.57%**	**2879.80**	**154.00**
	Random Sparsified	3.595e9	7.217e8	20.08%	1.195e9	1.516e8	12.69%	2901.60	**154.00**
	Sparsified Only	3.658e9	7.484e8	20.46%	1.217e9	1.535e8	12.61%	**2879.80**	**154.00**
$\gamma = \{1.1836, 1.2075, 1.2269\}$ @ $\{100k, 500k, 1M\}$									
100K	SPIN	**3.136e9**	**4.669e8**	**14.89%**	**1.098e9**	**1.229e8**	11.19%	**2922.60**	**160.40**
	Random Sparsified	3.140e9	4.681e8	14.91%	1.123e9	1.249e8	**11.12%**	2932.70	160.50
	Sparsified Only	3.192e9	4.814e8	15.08%	1.145e9	1.298e8	11.33%	**2922.60**	**160.40**
500K	SPIN	**3.584e9**	**6.564e8**	**18.32%**	**1.235e9**	**1.499e8**	12.13%	**3120.80**	**165.20**
	Random Sparsified	3.607e9	6.634e8	18.39%	1.241e9	1.523e8	12.28%	3137.60	**165.20**
	Sparsified Only	3.681e9	6.843e8	18.59%	1.277e9	1.524e8	**11.93%**	**3120.80**	**165.20**
1M	SPIN	**3.853e9**	**7.822e8**	**20.30%**	**1.264e9**	1.681e8	13.30%	**3127.60**	**168.70**
	Random Sparsified	3.880e9	7.877e8	**20.30%**	1.268e9	**1.670e8**	**13.17%**	3151.80	168.80
	Sparsified Only	3.950e9	8.180e8	20.71%	1.295e9	1.717e8	13.26%	**3127.60**	**168.70**
$\gamma = \{1.2754, 1.3112, 1.3404\}$ @ $\{100k, 500k, 1M\}$									
100K	SPIN	**3.347e9**	**5.055e8**	**15.10%**	**1.146e9**	**1.311e8**	**11.44%**	**3106.90**	**172.40**
	Random Sparsified	**3.340e9**	5.066e8	15.17%	1.176e9	1.353e8	11.51%	3115.50	**172.40**
	Sparsified Only	3.406e9	5.198e8	15.26%	1.193e9	1.369e8	11.47%	**3106.90**	**172.40**
500K	SPIN	**3.829e9**	**7.120e8**	**18.59%**	**1.292e9**	**1.640e8**	12.70%	**3336.20**	**178.00**
	Random Sparsified	3.856e9	7.236e8	18.77%	1.293e9	1.672e8	12.93%	3353.80	**178.00**
	Sparsified Only	3.938e9	7.439e8	18.89%	1.336e9	1.655e8	**12.38%**	**3336.20**	**178.00**
1M	SPIN	**4.119e9**	**8.462e8**	**20.55%**	**1.316e9**	**1.809e8**	13.74%	**3358.00**	**182.70**
	Random Sparsified	4.154e9	8.574e8	20.64%	1.329e9	1.829e8	13.76%	3389.40	182.90
	Sparsified Only	4.227e9	8.889e8	21.03%	1.361e9	1.860e8	**13.67%**	**3358.00**	**182.70**
$\gamma = \{1.3671, 1.4149, 1.4538\}$ @ $\{100k, 500k, 1M\}$									
100K	SPIN	**3.519e9**	5.461e8	15.52%	**1.195e9**	1.463e8	12.24%	**3274.10**	183.30
	Random Sparsified	3.534e9	**5.427e8**	**15.36%**	1.215e9	**1.432e8**	**11.79%**	3284.60	**183.40**
	Sparsified Only	3.590e9	5.607e8	15.62%	1.240e9	1.501e8	12.10%	**3274.10**	183.30
500K	SPIN	**4.098e9**	**7.779e8**	**18.98%**	1.344e9	**1.998e8**	14.39%	**3780.30**	**204.90**
	Random Sparsified	4.125e9	7.911e8	19.18%	**1.353e9**	**1.998e8**	14.49%	3798.80	204.80
	Sparsified Only	4.216e9	8.181e8	19.41%	1.388e9	2.016e8	**14.13%**	**3780.30**	**204.90**
1M	SPIN	**4.684e9**	**1.008e9**	**21.51%**	**1.414e9**	**2.198e8**	**15.54%**	**3835.70**	**211.70**
	Random Sparsified	4.723e9	1.024e9	21.68%	1.422e9	2.253e8	15.84%	3866.50	**211.70**
	Sparsified Only	4.811e9	1.066e9	22.16%	1.462e9	2.307e8	15.78%	**3835.70**	**211.70**
$\gamma = \{1.5507, 1.6224, 1.6807\}$ @ $\{100k, 500k, 1M\}$									
100K	SPIN	**7.520e9**	**6.660e8**	8.86%	**2.532e9**	**1.6045e8**	6.34%	**3630.40**	**207.10**
	Random Sparsified	7.545e9	6.695e8	8.87%	2.5905e9	1.6365e8	**6.32%**	3652.00	207.20
	Sparsified Only	7.655e9	6.975e8	9.11%	2.632e9	1.7015e8	6.46%	**3630.40**	**207.10**
500K	SPIN	**8.670e9**	**1.0055e9**	**11.60%**	**2.7615e9**	**2.2615e8**	**8.19%**	**3989.40**	**217.70**
	Random Sparsified	8.755e9	1.0345e9	11.81%	2.7735e9	2.3205e8	8.37%	4024.70	**217.70**
	Sparsified Only	8.955e9	1.1015e9	12.30%	2.851e9	2.389e8	8.38%	**3989.40**	**217.70**
1M	SPIN	**9.245e9**	**1.1965e9**	**12.94%**	**2.834e9**	**2.616e8**	**9.23%**	**4073.80**	226.60
	Random Sparsified	9.350e9	1.232e9	13.18%	2.8475e9	2.694e8	9.46%	4118.50	**226.50**
	Sparsified Only	9.530e9	1.310e9	13.75%	2.900e9	2.7745e8	9.57%	**4073.80**	226.60

quantities are conditioned on $\mathcal{R}$. Moreover, when we need to calculate path lengths between two nodes in the analysis below, we assume that the nodes are reachable from one to the other in the sparse graph. We define this event for the nodes u and v for the sparse graph $\widetilde{G}$ as

$$A_{u,v} = \{u \rightsquigarrow_{\widetilde{G}} v\}$$

Definition 5 (Multiplicative path stretch)**.** *Define the set $S_c := \{(u, v) : e = \{u, v\} \in E\}$ comprising the pair of nodes that have a direct edge in G. For any pair $(u, v) \in S_c$ define $D_{u,v} = 1$ as the length of the path between u and v in the dense graph. Similarly define $\widetilde{D}_{u,v}$ as the length of the path between u and v in the sparse graph, assuming that reachability is maintained between u and v in the sparse graph. The multiplicative stretch of $\widetilde{G}$ with respect to G is defined as*

$$\mathcal{S}(G, \widetilde{G}) := \frac{1}{m} \sum_{(u,v)\in S_c} \widetilde{D}_{u,v}$$

Definition 6 (Sparsification parameter)**.** *Let the probability of deleting an edge e in the dense graph during sparsification be $p_e := \lambda w_e$. Then, the sparsification parameter is defined as*

$$\alpha := \frac{1}{m} \sum_{e\in E} p_e$$

Definition 7 (Fallback stretch penalty)**.** *Fix $L \geq 2$. For each edge $e = \{u, v\} \in E$, choose k_e pairwise edge-disjoint replacement paths*

$$P_{e,1}, \ldots, P_{e,k_e} \subseteq G - e$$

from u to v, each of at most L hops. Define

$$q_{e,i} := 1 - \prod_{f\in P_{e,i}} (1 - p_f),$$

as the probability that path $P_{e,i}$ is destroyed by sparsification.

Let $BAD_{u,v}$ be the event that the edge e is deleted and all selected replacement paths are also cut. Assume that for some $B_{u,v} \geq L$,

$$\mathbb{E}\left[D_{u,v} \mid BAD_{u,v}, \mathcal{R}, A_{u,v}\right] \leq B_{u,v}.$$

Then, the fallback stretch penalty is defined as

$$\beta := \frac{1}{m} \sum_{e\in E} p_e (B_{u,v} - L) \prod_{i=1}^{k_e} q_{e,i}.$$

Theorem G.1. *The expected multiplicative stretch*

$$\mathbb{E}\Big[\mathcal{S}(G, \widetilde{G})\Big] = \frac{1}{m} \sum_{e=\{u,v\}\in E} \mathbb{E}\Big[\widetilde{D}_{u,v} \mid \mathcal{R}, A_{u,v}\Big] \leq 1 + (L-1)\alpha + \beta$$

Proof. Note that we calculate the expected stretch only among nodes that are reachable, i.e., $\widetilde{D}_{u,v} < \infty$. Obviously, if we include pairs of nodes that are pairwise unreachable, then the expected stretch becomes infinite. But by conditioning on $\mathcal{R}$, only a negligible fraction of nodes become pairwise unreachable in the sparse graph, if they were already not unreachable in the dense graph.

Fix some edge $e = \{u, v\}$. If the edge is retained after sparsification then $\widetilde{D}_{u,v} = 1$. Otherwise, if the edge is deleted and one of the remaining replacements paths of length $\leq L$ are retained, then $\widetilde{D}_{u,v} \leq L - 1$. Finally, if none of the replacement edges are retained, then we revert to a fallback path of length $B_{u,v}$. Then, we have

$$\mathbb{E}\Big[\widetilde{D}_{u,v} \mid \mathcal{R}, A_{u,v}\Big] \leq 1 + (L-1)\cdot\mathbb{P}(\text{edge } e \text{ is deleted} \mid \mathcal{R}, A_{u,v}) + (B_{u,v} - L)\cdot\mathbb{P}(BAD_{u,v} \mid \mathcal{R}, A_{u,v})$$

Now, we know that $\mathcal{R}$ and $A_{u,v}$ is monotone increasing with respect to edge additions, and the event that edge e is cut is monotone decreasing with respect to edge deletions. Thus, by invoking the FKG inequality, $\mathbb{P}\left[\text{edge } e \text{ is deleted} \wedge \mathcal{R} \wedge A_{u,v}\right] \leq \mathbb{P}\left[\text{edge } e \text{ is deleted}\right] \mathbb{P}\left[\mathcal{R} \wedge A_{u,v}\right]$. This implies $\mathbb{P}(\text{edge } e \text{ is deleted} \mid \mathcal{R}, A_{u,v}) \leq \mathbb{P}\left[\text{edge } e \text{ is deleted}\right]$. We can similarly derive $\mathbb{P}(BAD \mid \mathcal{R}, A_{u,v}) \leq \mathbb{P}\left[BAD\right]$.

Because the replacement paths are pairwise edge-disjoint and none contains e, $\mathbb{P}\left[BAD\right] = p_e \prod_{i=1}^{k_e} q_{e,i}$. Thus,

$$\mathbb{E}\Big[\widetilde{D}_{u,v} \mid \mathcal{R}, A_{u,v}\Big] \leq 1 + (L-1)\cdot p_e + (B_{u,v} - L)p_e \prod_{i=1}^{k_e} q_{e,i}$$

By linearity of expectation,

$$\mathbb{E}\Big[\mathcal{S}(G,\widetilde{G})\Big] = \mathbb{E}\left[\frac{1}{m}\sum_{e\in E}\widetilde{D}_{u,v} \mid \mathcal{R}, A_{u,v}\right] = \frac{1}{m}\sum_{e\in E}\mathbb{E}\Big[\widetilde{D}_{u,v} \mid \mathcal{R}, A_{u,v}\Big]$$

$$\leq 1 + \frac{1}{m}(L-1)\cdot\sum_{e\in E} p_e + \frac{1}{m}\cdot\sum_{e\in E}(B_{u,v}-L)p_e\prod_{i=1}^{k_e} q_{e,i} = 1 + (L-1)\alpha + \beta$$

□

Assumption 1 (Stretch-weighted query concentration)**.** *Let $\mathcal{Q}$ be a query distribution. Let P_q be the path associated with the candidate returned by dense graph G and let $h_q := |P_q|$. Define*

$$a_e := \mathbb{E}\left[\frac{\mathbf{1}\{e \in P_q\}}{h_q}\right] \quad \forall e \in E$$

Then, the stretch-weighted query concentration parameter is defined as

$$\chi := \frac{m\sum\limits_{e=\{u,v\}} a_e\mathbb{E}\left[\widetilde{D}_{u,v} - 1 \mid \mathcal{R}, A_{u,v}\right]}{\sum\limits_{e=\{u,v\}}\mathbb{E}\left[\widetilde{D}_{u,v} - 1 \mid \mathcal{R}, A_{u,v}\right]}$$

Theorem G.2. *Let the ef be the ef-search value for the dense graph G. Fix some value $\gamma \in [1, \mathbb{E}\Big[\mathcal{S}(G,\widetilde{G})\Big]]$ and let $\delta \in [1, \gamma_{\max}] \to (1,\infty)$ be a non-decreasing function. Let x_q be the candidate returned by the nearest-neighbor search in G with candidate set size ef, and $\widetilde{x}_q$ be the candidate returned during the search in $\widetilde{G}$ with candidate set size $\lceil\gamma\cdot ef\rceil$, for a given query $q \in \mathcal{Q}$. Then,*

$$\mathbb{P}(d(q,\widetilde{x}_q^{(\gamma)}) \leq d(q,x_q) \mid \mathcal{R}) \geq 1 - \frac{\chi\cdot\mathbb{E}\Big[\mathcal{S}(G,\widetilde{G})\Big]}{\delta(\gamma)} - \rho_\gamma$$

where ρ_γ is the probability that the x_q was not examined during the search in the sparse graph, even though the search budget $\lceil\gamma\cdot ef\rceil$ was sufficient.

Proof. For the query q, let $\mathcal{E}_q$ be the event that the candidate returned by the dense graph x_q is also reachable in the sparse graph from the random starting node. Also, let P_q be the path associated with x_q returned by the dense graph search.

$$X_q := \frac{1}{|P_q|}\sum_{e=\{u,v\}\in P_q}\widetilde{D}_{u,v}.$$

For the multiplier γ, let $\delta(\gamma)$ be the amount of path stretch due to sparsification that the increase in search budget can tolerate. Let BAD_γ be the event that x_q was not examined during the search in the sparse graph, despite $X_q \leq 1 + \delta(\gamma)$. Then define,

$$\rho_\gamma := \mathbb{P}\left[BAD_\gamma\right] = \mathbb{P}(x_q \text{ is not examined} \mid \mathcal{E}_q, X_q \leq 1 + \delta(\gamma))$$

Next, we show that $\mathbb{E}[X_q - 1 \mid \mathcal{R}, \mathcal{E}_q] \leq \chi \cdot \mathbb{E}\Big[\mathcal{S}(G, \widetilde{G}) - 1\Big]$. This follows from linearity of expectation since

$$\mathbb{E}[X_q - 1 \mid \mathcal{R}, \mathcal{E}_q] = \sum_{e=\{u,v\}\in E} a_e \mathbb{E}\Big[\widetilde{D}_{u,v} - 1 \mid \mathcal{R}, A_{u,v}\Big] = \frac{\chi}{m} \sum_{e=\{u,v\}\in E} \mathbb{E}\Big[\widetilde{D}_{u,v} - 1 \mid \mathcal{R}, A_{u,v}\Big] \leq \chi \cdot \mathbb{E}\Big[\mathcal{S}(G, \widetilde{G}) - 1\Big]$$

If the search on the sparse graph returns $\widetilde{x}_q^{(\gamma)}$, such that $d(q, \widetilde{x}_q^{(\gamma)}) > d(q, x_q)$, then x_q was not examined duing the search. This happens when either $\{X_q > \delta(\gamma)\}$ or the event BAD_γ. By Markov's inequality $\mathbb{P}(X_q > 1 + \delta(\gamma) \mid \mathcal{R}, \mathcal{E}_q) \leq \frac{\chi \cdot \mathbb{E}\left[\mathcal{S}(G,\widetilde{G}) - 1\right]}{\delta(\gamma)}$. Thus, a union bound over the events and taking the complement gives the result. □

G.2 IDEAL-CACHE ANALYSIS

We analyze the cache complexity of the sparse graph produced by SPIN using the ideal-cache model. Assume that the cache has capacity M and cache-line (block) size B. The cache is fully associative and uses an optimal replacement policy. Data is transferred between main memory and cache in blocks of size B. Let the sparse graph be partitioned into clusters $\mathcal{C} = \{C_1, C_2, \ldots\}$. We assume that the graph data associated with every cluster fits entirely in cache. In particular, if $s(C)$ denotes the memory footprint of cluster C, then for every $C \in \mathcal{C}$ $s(C) \leq M$. We first provide some important definitions that will be used in the analysis.

Definition 8 (Workload expansion factor)**.** *Fix some $\gamma > 1$. Define $\mathcal{H}_{G,\mathcal{Q}}(ef)$ the expected number of nodes expanded during a search in the dense graph for a query $q \sim \mathcal{Q}$ when the search candidate set size is ef. Similarly, define $\mathcal{H}_{\widetilde{G},\mathcal{Q}}(\gamma ef)$ as the expected number of nodes expanded during search in the sparse graph with the search candidate set size $\lceil \gamma ef \rceil$. Then, define workload expansion factor as*

$$\omega_{\widetilde{G},\mathcal{Q}}(\gamma) := \frac{\mathcal{H}_{\widetilde{G},\mathcal{Q}}(\gamma ef)}{\mathcal{H}_{G,\mathcal{Q}}(ef)}$$

Definition 9 (Cross-cluster transitions)**.** *Given a sparse graph $\widetilde{G}$ that is clustered and reordered by* SPIN*, define the maximum number of cache lines occupied by a cluster as C_{max}. Also define $T_q(\gamma)$ as the number cross cluster edge transitions when serving a query $q \in \mathcal{Q}$.*

Definition 10. *For a query q, let $v_1(q), v_2(q), \ldots, v_{H_q}(q)$ denote the sequence of vertices expanded during the search, where H_q is the number of node expansions. Let $C(v)$ denotethe the cluster containing vertex v. We define the per query cross-cluster transitions T_q as*

$$T_q = \sum_{t=2}^{H_q} \mathbf{1}\left\{C(v_t(q)) \neq C(v_{t-1}(q))\right\}.$$

Theorem G.3 (Ideal-cache bound)**.** *Suppose that the graph data associated with each cluster occupies at most $S_{\max} \leq M$ bytes and is stored contiguously in memory. Then the number of cache misses, Q_q, incurred by query q satisfies*

$$Q_q \in \mathrm{O}\left(\frac{S_{\max}}{B}(T_q + 1)\right).$$

Proof. Consider a single maximal run of expansions inside some cluster C. Since the entire graph representation of a cluster $C_i \in \mathcal{C}$ occupies at most $S_{\max}$ bytes, it spans at most $\mathrm{O}\big(\frac{S_{\max}}{B}\big)$ cache blocks. Because $s(C) \leq M$, all blocks belonging to C can simultaneously fit in the ideal cache.

Therefore, during one maximal run inside C, each block of C needs to be fetched from main memory at most once. Hence, the cache cost of a single cluster run is at most $\mathrm{O}\big(\frac{S_{\max}}{B}\big)$

Define R_q as a *cluster run* – maximal consecutive subsequence of expansions belonging to the same cluster. Then, there are $R_q = T_q + 1$ cluster runs during query q. It follows that

$$Q_q = \mathrm{O}\left(\frac{S_{\max}}{B} R_q\right) = \mathrm{O}\left(\frac{S_{\max}}{B}(T_q + 1)\right).$$

□

Theorem G.4. *For a given query distribution $\mathcal{Q}$, let*

$$\tau_{\mathcal{Q}} = \frac{\mathbb{E}_{q\sim\mathcal{Q}}[T_q]}{\mathbb{E}_{q\sim\mathcal{Q}}[H_q - 1]}$$

, where H_q is the number of node expansions for the query. Define the expected number of graph related cache misses during queries on the sparse, clustered graph produced by SPIN*, $M_{\widetilde{G}}$. Then,*

$$M_{\widetilde{G}} = \mathrm{O}\Big(1 + \tau_q \cdot \mathcal{H}_{\widetilde{G},\mathcal{Q}}(\gamma ef)\Big)$$

Proof. Taking expectations in $Q_q = \mathrm{O}\big(\frac{S_{\max}}{B}(T_q + 1)\big)$

$$M_{\widetilde{G}} = \mathrm{O}\left(\frac{S_{\max}}{B}\left(1 + \mathbb{E}[T_q]\right)\right)$$

Using $\mathbb{E}[T_q] = \tau_{\mathcal{Q}}\left(\mathbb{E}[H_q] - 1\right)$

$$M_{\widetilde{G}} = \mathrm{O}\left(\frac{S_{\max}}{B}\left(1 + \tau_{\mathcal{Q}}\mathcal{H}_{\widetilde{G},\mathcal{Q}}(\gamma ef)\right)\right)$$

We get the result by assuming that the hardware parameters are fixed $\frac{S_{\max}}{B} = \mathrm{O}(1)$. □