LEANN: A LOW-STORAGE OVERHEAD VECTOR INDEX

Yichuan Wang[† 1] **Zhifei Li**[1] **Shu Liu**[1] **Yongji Wu**[† 1] **Ziming Mao**[1] **Yilong Zhao**[1] **Xiao Yan**[2] **Zhiying Xu**[* 3]
Yang Zhou[1 4] **Ion Stoica**[1] **Sewon Min**[1] **Matei Zaharia**[1] **Joseph E. Gonzalez**[1]

ABSTRACT

Embedding-based vector search underpins many important applications, such as recommendation and retrieval-augmented generation (RAG). It relies on vector indices to enable efficient search. However, these indices require storing high-dimensional embeddings and large index metadata, whose total size can be several times larger than the original data (e.g., text chunks). Such high storage overhead makes it difficult, or even impractical, to deploy vector search on personal devices or large-scale datasets. To tackle this problem, we propose LEANN, a storage-efficient index for vector search that recomputes embeddings on the fly instead of storing them, and compresses state-of-the-art proximity graph indices while preserving search accuracy. LEANN delivers high-quality vector search while using only a fraction of the storage (e.g., 5% of the original data) and supporting storage-efficient index construction and updates. On real-world benchmarks, LEANN reduces index size by up to 50× compared with conventional indices, while maintaining SOTA accuracy and comparable latency for RAG applications.

1 INTRODUCTION

Advances in foundation models have led to increasingly powerful embedding models, and *embedding-based vector search* has become a core functionality underpinning many important applications, such as content search (Lee et al., 2024; Yin et al., 2024), personal assistants (He et al., 2019; Cai et al., 2024), and question answering (Yang et al., 2018; Joshi et al., 2017). In particular, data objects with complex semantics (e.g., texts, images, videos) are mapped to high-dimensional vectors with an embedding model, so that semantically similar or related objects have a small distance between their embeddings. To retrieve objects from a database, the query object (e.g., a text description) is first embedded as a query vector and then used to search for the top-k most similar vectors. Since exact vector search requires a linear scan in high-dimensional space, *approximate nearest neighbor search* (ANNS) is commonly adopted (Aumüller et al., 2020), which returns most rather than all of the top-k neighbors. The result quality of ANNS is typically measured by *recall*, defined as the fraction of ground-truth top-k neighbors that appear in the k retrieved vectors.

Table 1 shows that when retrieval-augmented generation (RAG) is applied to a question answering (QA) dataset, vector search methods such as *Hierarchical Navigable Small World* (HNSW) (Malkov & Yashunin, 2018) yield substantially higher downstream accuracy than traditional keyword-based search approaches like BM25 (Craswell et al., 2021). This is because vector search better retrieves passages that are semantically related to the query intent.

Table 1. Storage overhead and runtime statistics of different indexing methods for RAG, evaluated on a **76 GB** text datastore (Computer, 2023) and a QA dataset (Kwiatkowski et al., 2019) using the *Qwen3-4B* model on an RTX 4090.

Metrics	BM25	HNSW	PQ	LEANN
Downstream accuracy (%)	*18.3*	***25.5***	*17.9*	***25.5***
Storage size (GB)	*59*	*188*	*20*	***4***
Index metadata	-	*15*	*15*	***2***
Vectors	-	*173*	*5*	***2***
End-to-end latency (s)	*21.36*	*20.95*	*25.45*	***23.34***
Search	*0.03*	*0.05*	*4.53*	*2.48*
Response generation	*21.33*	*20.90*	*20.92*	*20.86*

Deploying ANNS: Challenges and Opportunities. Vector search demands substantial storage, as high-dimensional embeddings and index metadata can be several times larger than the original data (Shao et al., 2024). Table 1 shows that a 76 GB text corpus requires 173 GB for embeddings and 15 GB for the HNSW index, a state-of-the-art graph-based ANN method, thereby more than doubling the data size. This imposes a high storage burden for many use cases, including RAG workloads on personal devices and semantic search over large datasets (e.g., logs or documents). In particular, running vector search locally (e.g., on laptops or workstations) is attractive because it preserves privacy

Under Review. *This work does not relate to the position at Amazon. † Corresponding authors. Email: yichuan_wang@berkeley.edu, wuyongji317@gmail.com. [1]UC Berkeley [2]CUHK [3]Amazon Web Services [4]UC Davis. Correspondence to: Yichuan Wang <yichuan_wang@berkeley.edu>, Yongji Wu <wuyongji317@gmail.com>.

and enables offline access without uploading data to the cloud (Wang & Chau, 2024). However, the storage capacity of personal devices is often insufficient for large-scale embeddings and indices.

To reduce storage overhead, a common approach is to compress embeddings using lossy vector quantization methods such as product quantization (PQ) (Jégou et al., 2011). Approximate distances can then be computed between the query and the compressed vectors. However, achieving small vector sizes requires a high compression ratio. For example, PQ needs about 35× compression to reduce the vectors to 5 GB in Table1. At such a high ratio, large quantization errors degrade the downstream accuracy of vector search to levels even below keyword search with BM25. Moreover, the 15 GB HNSW index cannot be compressed using vector quantization and still burdens personal devices.

An important observation from Table 1 is that in RAG workloads, LLM generation dominates end-to-end latency (i.e., response generation takes over 20s on an RTX 4090, while vector search completes in milliseconds). This long generation time, common in complex reasoning or agentic tasks, relaxes the strict requirement for search latency. Since overall latency is bounded by generation, we can trade a small amount of search latency for substantial storage savings, enabling much more compact vector indexes. This is an attractive trade-off for personal devices or resource-constrained deployments. Motivated by this observation, we ask:

> *Can we design a vector index that dramatically reduces storage overhead while maintaining search accuracy and meeting reasonably relaxed latency requirements?*

Our solution LEANN. We present LEANN as a vector index tailored for storage-constrained environments with both system and algorithmic optimizations. LEANN can reduce the index footprint to below 5% of the original data while preserving high result accuracy and reasonable retrieval latency. At its core, LEANN is guided by two insights:

The first insight is that state-of-the-art proximity graph indexes (e.g., HNSW, which we build upon) require each query to visit only a small subset of embeddings to find its nearest neighbors. Thus, instead of storing all embedding vectors, LEANN recomputes them at query time using the same encoder as in index building. However, naive embedding recomputation can lead to significant latency overhead. To mitigate this, LEANN introduces a two-level search algorithm that uses the inaccurate approximate distances at high compression ratios to prune embedding recomputations. Moreover, LEANN also employs a dynamic batching mechanism that aggregates embedding computations across search hops over the proximity graph index to improve GPU utilization and reduce recomputation latency.

While embedding recomputation allows removing exact vectors, the proximity graph index metadata can still be large, as shown in Table 1. For example, if a node (i.e., vector) has 64 neighbors, each adjacency list takes 256 bytes, which is already 25% in size over the typical 1 KB document chunk for original data (Shao et al., 2024). Our second insight is that the high-degree nodes in a proximity graph are visited much more frequently than the low-degree nodes and thus are more important for vector search. Hence, LEANN applies a high-degree preserving graph pruning strategy, which removes the low utility edges of low-degree nodes while preserving the edges of high-degree "hub" nodes (Munyampirwa et al., 2024). This substantially reduces index size without sacrificing search accuracy and efficiency.

Besides the two key designs, LEANN incorporates a storage-efficient sharded merging pipeline index building strategy, which ensures that storage consumption never exceeds a small budget even when building the index for a large dataset. In addition, LEANN also supports updating the compressed index (e.g., adding new data). This significantly reduces the update time while remaining storage-efficient.

We implement LEANN[1] on top of FAISS (Douze et al., 2025), one of the most popular frameworks for ANNS, and evaluate it across four information retrieval (IR) benchmarks and beyond. Our experiment platforms include a server with NVIDIA RTX 4090 GPU (NVIDIA, 2022) and an M1-based Mac (AWS, 2023). The results show that LEANN reduces storage consumption by more than 50× compared to state-of-the-art vector indexes while maintaining result accuracy. When applied to RAG tasks, LEANN incurs about 10% end-to-end latency overhead (see Table 1).

To summarize, we make the following contributions:

- We are the first to study the storage challenge of vector search and design LEANN, a novel storage-efficient index that performs on-the-fly embedding recomputation and applies high-degree preserving graph pruning to reduce storage while preserving accuracy.
- We incorporate a suite of optimizations, including two-level search, dynamic batching, storage-constrained index building, and efficient index update, to make the entire pipeline both fast and storage efficient.
- We demonstrate that LEANN achieves over 90% top-3 recall within one second while using less than 5% of the raw data storage, maintaining comparable latency for RAG workloads.

[1]Code repository: `https://github.com/yichuan-w/LEANN`.

2 BACKGROUND

Vector search. To retrieve semantically related or similar objects for unstructured data (e.g., texts, images, videos), vector search is widely used. In particular, given a vector dataset $\mathcal{X} = \{x_1, x_2, \cdots, x_N\} \subset \mathbb{R}^d$ and a query vector $q \in \mathbb{R}^d$, vector search finds the top-k vectors in $\mathcal{X}$ that are the most similar to q, i.e.,

$$|\mathcal{S}_q| = k \text{ with } \|q - x_i\| \leq \|q - x_j\| \; \forall x_i \in \mathcal{S}_q, x_j \in \mathcal{X} \setminus \mathcal{S}_q. \quad (1)$$

The similarity function can also be the inner product or cosine similarity, where larger values indicate higher similarity. However, due to the curse of dimensionality in high-dimensional spaces, exact vector search requires a linear scan (Wang et al., 2021), which is costly for large datasets. As such, approximate nearest neighbor search (ANNS) is commonly used (Malkov & Yashunin, 2018; Lempitsky, 2012), which trades minor result inaccuracies for substantially lower query latency. The result quality of ANNS is usually measured by *recall*, which is the fraction of ground-truth top-k neighbors that are contained in the set $\mathcal{S}'_q$ of returned approximate neighbors, i.e.,

$$Recall@K = |\mathcal{S}_q \cap \mathcal{S}'_q|/k. \quad (2)$$

Applications such as RAG typically require a high recall (e.g., ≥ 0.9) for good performance (Shen et al., 2024).

Indexes are essential for the efficiency of vector search by confining the distance computations to a small portion of vectors. The storage cost of a vector index consists of two components, i.e., the vectors and the index metadata. Two types of vector indexes are the most popular, i.e., IVF (Lempitsky, 2012)) and proximity graph (Malkov & Yashunin, 2018; Fu et al., 2019; Subramanya et al., 2019). IVF groups the vectors into clusters and represents each cluster with a center vector, and a query first scans the centers and then checks the vectors in a few most similar clusters. Proximity graph connects similar vectors to form a graph and conducts vector search via a best-first traversal on the graph. Although IVF is cheaper to build and requires smaller space to store the index structure, proximity graph achieves the SOTA efficiency for vector search in that it requires much fewer distance computations (Subramanya et al., 2019).

Best-first search on proximity graph. The variants of proximity graph index (e.g., HNSW (Malkov & Yashunin, 2018), NSG (Fu et al., 2019), Vamana (Subramanya et al., 2019)) differ in their edge connection rules, but the query processing algorithm is similar. Algorithm 1 illustrates the best-first search on the proximity graph. The search maintains a bounded priority queue C of candidate nodes, ordered by their distances to the query q. At each exploration step (Line 5), the algorithm *reads* (but does not remove) the closest unvisited node u from C and explores its neighbors. For each neighbor whose distance has not been computed,

Algorithm 1 Best-First Search on Graph-based Index

1: **Input:** Graph G, query q, entry point p, result count k, queue size ef
2: **Output:** k nearest neighbors to q
3: Init size-ef priority queue C with $(p, \text{Dist}(q, x_p))$
4: **while** C has unvisited node **do**
5: Read the closest but unvisited node u in C
6: Mark u as visited
7: **for** each neighbor v of u in G **do**
8: **if** $\text{Dist}(q, x_v)$ is not computed **then**
9: Extract embedding x_v for v
10: Try to insert $(v, \text{Dist}(q, x_v))$ into C
11: **return** The k nodes with the smallest distances in C

the algorithm extracts the embedding, computes its distance to q, and inserts the neighbor into C if the queue is not full or if the neighbor is closer than the tail entry of C. The parameter ef bounds the queue size and acts as a *quality knob*: a larger ef improves recall at the cost of more distance computations. The search terminates once all the nodes in C have been visited. Empirically, graph-based indexes achieve high recall with only $\mathcal{O}(\log N)$ embedding extractions and distance computations. This is because the graph traversal can quickly converge on the neighbors of the query by moving to more similar neighbors in each step.

3 LEANN OVERVIEW

Figure 1 shows the end-to-end workflow of LEANN, which includes offline index construction and online query serving.

Offline stage Given a dataset of items, such as chunked unstructured text, LEANN computes embeddings and builds a graph-based vector index. To minimize storage, it applies a graph pruning algorithm that preserves high-degree nodes (§5) and discards dense embeddings, retaining only the pruned graph structure. During construction, LEANN also builds a lightweight product quantization (PQ) table that stores approximate embeddings for fast distance estimation during query processing (§4). Optionally, if a peak storage budget is specified, LEANN adopts a graph partitioning-based build strategy (§6) to keep the storage footprint within this bound by constructing and merging shards sequentially. At the end of the offline stage, LEANN persists two compact components: (i) the pruned graph adjacency lists and (ii) the PQ-compressed embedding table.

Online stage. When a query arrives, LEANN searches over the pruned graph using Algorithm 1. To accelerate query processing, LEANN employs a two-level search strategy that first computes lightweight approximate distances using PQ embeddings and then recomputes exact embeddings on demand for the most promising candidates via the local embedding generator. During recomputation, LEANN ap-

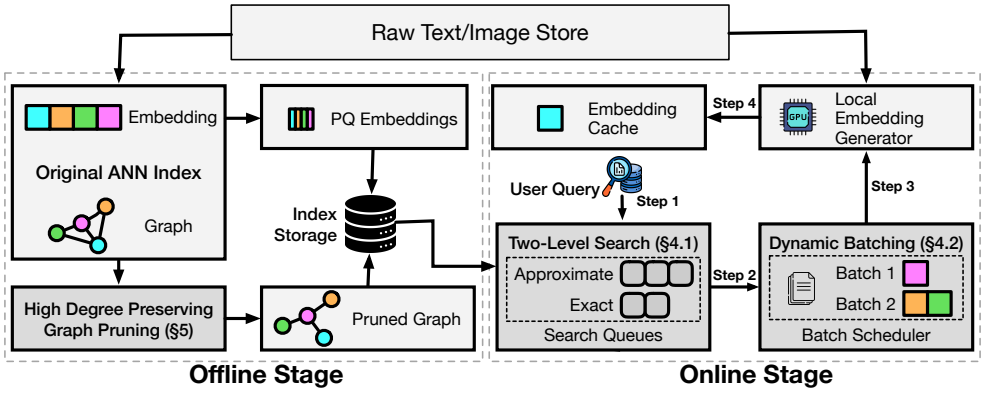


Figure 1. LEANN System Diagram. The system combines high-degree preserving graph pruning for minimal storage footprint with graph-based recomputation and two-level search with dynamic batching for efficient query processing (Steps 1-4).

plies dynamic batching to group multiple candidate nodes across exploration steps, improving GPU utilization and reducing end-to-end latency. Finally, the system ranks all visited nodes by their exact distance to the query and returns the top results to the downstream task. LEANN also provides a lightweight update pipeline for dynamic index maintenance (§6) and, if disk capacity allows, an optional embedding cache to store frequently accessed nodes and avoid redundant recomputation.

Storage composition. Across both stages, LEANN stores compact structures. For N data chunks (nodes), the pruned graph requires $O(N \times |D|)$ integer entries, where $|D|$ denotes the average node degree. The PQ table employs a $100\times$ smaller codebook than the original FP32 embeddings, occupying $O(4N \times dim/100)$ bytes (e.g., $dim = 768$). Together, these components reduce storage by up to $50\times$ compared to conventional dense indexes.

Use cases. LEANN can conduct vector search on user devices (e.g., laptops and personal servers), on which storage is highly limited. Our experiment evaluation also focuses on this use case. LEANN may also be used for data lakes, which contain many datasets, and some cold datasets are queried infrequently (Mageirakos et al., 2025). Storing indexes for these cold datasets incurs high space overheads, while recomputing embeddings for them is inexpensive due to low query frequency. Similarly, LEANN can handle datasets whose embeddings have skewed access patterns, e.g., for recommendation and content search, popular entries are more likely to become the results of vector search (Mohoney et al., 2023). For these datasets, LEANN may store exact vectors for the popular entries and use embedding recomputation for the cold entries to reduce storage.

Algorithm 2 Two-Level Search

Input: query q, entry point p, re-ranking ratio α, result size k, search queue length ef
Output: k closest neighbors to q
Init size-ef priority queue EQ with $(p, \text{Dist}(q, x_p))$
Init empty approximate priority queue AQ
while EQ has unvisited node **do**
  Read the closest unvisited node u from EQ
  Mark u as visited
  for each neighbor v of u **do**
    if approximate distance to q not computed **then**
      Extract approximate embedding $\tilde{x}_v$ for v
      Insert $(v, \text{Dist}(q, \tilde{x}_v))$ into AQ
  $C \leftarrow$ top $\alpha\%$ candidates in AQ, excluding EQ
  for each $c \in C$ **do**
    Recompute embedding x_c
    Try to insert $(c, \text{Dist}(q, x_c))$ into EQ
return The k nodes with smallest distances in EQ

4 Graph-Based Recomputation

In this section, we introduce our efficient recomputation pipeline, which reduces the number of nodes involved in recomputation (§4.1) and fully utilizes GPU resources during the process (§4.2).

4.1 Two-Level Search with Hybrid Distance

Motivation. LEANN stores PQ codes for all vectors to enable approximate distance computation. Existing systems such as DiskANN search the proximity graph using these approximate distances and then re-rank the top candidates with exact distances, e.g., re-ranking the top-100 approximate neighbors for top-10 results. However, this approach is problematic for LEANN because our PQ codes use a high compression ratio for compact storage, leading to large quantization errors. In particular, the approximate

distances can lead the graph traversal to detours by visiting sub-optimal candidates, which prolongs search time. Moreover, some ground-truth neighbors may be missed due to sub-optimal candidates, and re-ranking more approximate neighbors will not improve recall in this case (see Figure 4). To tackle this problem, we interleave approximate and exact distance computations rather than isolating them as in existing systems. Specifically, we use exact distances to select candidates to visit so that the graph traversal maintains high quality, while approximate distances are used to prune unnecessary exact computations, achieving accuracy and efficiency at the same time.

Solution. Algorithm 2 outlines the complete procedure. At each exploration step, LEANN first computes approximate distances for all neighbors using PQ (Line 11) and maintains an approximate queue (AQ) that stores these values for all explored nodes. Instead of recomputing every neighbor's embedding, we define a re-ranking ratio α and extract the top $\alpha\%$ of nodes from AQ, excluding those already in the exact queue (EQ). The selected subset C (Line 12) is then recomputed exactly, and each node is inserted into EQ for further exploration.[2] This hybrid strategy significantly reduces recomputation without sacrificing accuracy.

Discussion. In practice, LEANN uses a PQ table with a 100× smaller codebook, representing embeddings in $O(4N \times dim/100)$ bytes (with $dim = 768$ in our setup). These approximate distances provide an inexpensive yet effective signal for early filtering, and recomputation is reserved only for a small fraction of top-ranked candidates. Although PQ introduces quantization errors, selective exact recomputation restores ranking fidelity and ensures retrieval quality. The method generalizes easily to other forms of approximation, such as using distilled embedding models or link-and-code representations (Douze et al., 2018).

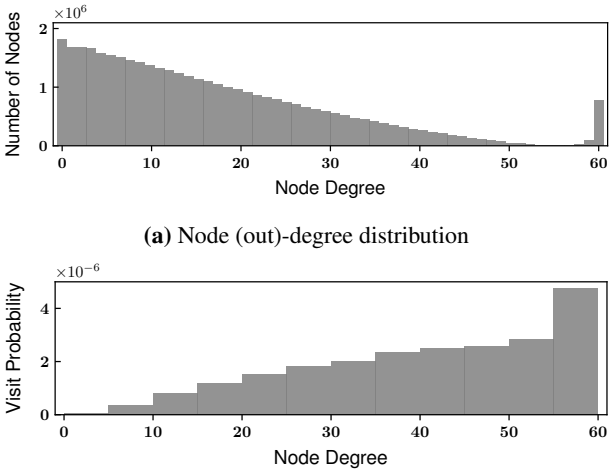
4.2 Dynamic Batching for Recomputation

Motivation. In the naive approach, embeddings are recomputed one by one for each neighbor node, as shown in Line 14 of Algorithm 2. To better utilize the GPU, LEANN batches all neighbor nodes within an exploration step so their embeddings are recomputed together. However, even with this optimization, each batch remains small—limited by the degree of the current node u. The problem becomes more pronounced in the two-level search algorithm (Line 12), where the candidate set per step is even smaller.

Solution. To further improve GPU utilization, LEANN introduces a dynamic batching strategy that relaxes the strict data dependency in best-first search (Algorithm 1). While this introduces slight staleness in the exploration order, it enables batching across multiple exploration steps, increasing the effective batch size and improving throughput.

Specifically, LEANN dynamically collects a group of the closest candidates from the priority queue. The algorithm accumulates nodes requiring recomputation until a target batch size is reached (e.g., 64), which can be efficiently determined through lightweight offline profiling. This dynamic batching mechanism integrates naturally with the two-level search strategy described in §4.1: in practice, LEANN accumulates nodes in the set C across iterations until the predefined batch threshold is reached, then recomputes embeddings for all nodes in C together.

This dynamic batching approach allows LEANN to batch nodes across multiple graph exploration steps regardless of individual node degrees, trading off slight staleness for significantly improved GPU utilization compared to single-step processing.



(a) Node (out)-degree distribution

(b) Node access probability.

Figure 2. HNSW graph analysis reveals skewed access and degree distributions, with node degrees capped at 60 by HNSW.

5 Compact Graph Structure

With the two-level search and dynamic batching mechanisms optimizing recomputation latency, we now examine how LEANN further reduces storage overhead in graph index metadata through a high-degree preserving graph pruning algorithm. As noted in §3, although LEANN eliminates the need to store exact embeddings by recomputing them at query time, the graph metadata that guides the search still incurs significant storage cost (see Table 1). In fact, even with embeddings, the index metadata alone can exceed 30%

[2] AQ tracks all previously visited nodes, allowing LEANN to revisit earlier neighbors that become more promising as the search progresses.

of the total storage (Severo et al., 2025).

Problem Formulation. Given a disk usage constraint B, LEANN aims to prune the graph index so that the metadata storage remains within budget while maintaining retrieval accuracy. Formally, the optimization problem is:

$$\begin{aligned} \min \quad & T(G_1) &&= \sum_{i=1}^{ef} |V_i| \\ \text{s.t.} \quad & \text{Space}(G_1) &&= \sum_{v \in V(G_1)} \deg(v)\, s_{\text{edge}} \leq B, \\ & \text{Acc}(G_1) &&\geq \tau \end{aligned} \tag{3}$$

Here, G_1 is the pruned graph, and $|V_i|$ is the number of nodes recomputed in each exploration step during search using G_1. A smaller $T(G_1)$ indicates fewer recomputations and thus lower query latency. Space(G_1) denotes the metadata size of the graph, stored in a compressed sparse row (CSR) format, which records each node's outgoing neighbor IDs. $\deg(v)$ is the out-degree of node v, and each stored neighbor ID takes 4 bytes. The goal is to minimize recomputation cost while keeping the graph within the storage budget B and accuracy (recall) above the threshold τ.

Motivation. There are two naive ways to shrink the proximity graph index: (1) randomly removing edges, and (2) lowering the degree limit for each node. Both approaches significantly degrade search accuracy even under mild size reductions, as shown in Figure 6, because they harm graph connectivity, which is crucial for effective traversal. From Figure 2, we observe that the edges are not equally important: a small fraction of nodes have high degrees (i.e., approaching or at the degree limit), and these nodes are accessed much more frequently than the low-degree nodes. These high-degree nodes essentially serve as the "navigation hubs" for graph traversal, and similar phenomena are also observed in (Munyampirwa et al., 2024). As such, we preserve the edges for the high-degree nodes to ensure good navigability of the proximity graph and conduct edge pruning for the low-degree nodes.

Solution: Our key insight is that preserving a small set of hub nodes is sufficient to maintain search performance. Following prior work (Ren et al., 2020; Munyampirwa et al., 2024), high-degree nodes serve as the backbone of the graph's connectivity; thus, LEANN focuses on retaining these hubs while reducing the overall number of edges. Algorithm 3 outlines this high-degree preserving graph pruning strategy.

We assign degree thresholds based on node importance: most nodes are limited to a lower degree m, while a small fraction of nodes ($\beta\%$) can retain up to M connections (Line 7). Empirically, we set $m = M/5$, which keeps a small local neighborhood for ordinary nodes. We determine

Algorithm 3 High-Degree Preserving Graph Pruning

1: **Input:** Original graph G with vertex set V; construction queue length efC; maximum degree M for high-degree nodes; lower degree m for others ($m < M$); proportion of high-degree nodes β
2: **Output:** Pruned graph G_1
3: Init $D[v] \leftarrow \deg(v)$ for all $v \in V$; $G_1 \leftarrow \emptyset$
4: $V^* \leftarrow$ nodes with the top $\beta\%$ highest degrees in D
5: **for** $v \in V(G)$ **do** ▷ Construct G_1
6: $W \leftarrow$ Search(v, efC) ▷ See Algorithm 1
7: **if** $v \in V^*$ **then** $M_0 \leftarrow M$
8: **else** $M_0 \leftarrow m$
9: Select up to M_0 neighbors from W using RNG-style pruning (see Appendix A)
10: Add bidirectional edges between v and neighbors
11: If $\deg(u) > M$ for any neighbor u, shrink to M with the same heuristic

M for a given storage budget B through offline profiling. The parameter m directly controls the aggressiveness of pruning: a smaller m reduces graph metadata size by limiting per-node degree, at the cost of reduced redundancy and potentially longer traversal paths. This reflects an explicit and interpretable storage–graph-quality trade-off. We use node degree as a proxy for node importance and select the top $\beta\%$ of nodes by degree (Line 4). For hub retention, we observe consistent behavior across datasets: setting β in the range of 3–5% yields a strong operating point, while larger values provide diminishing returns, indicating that a small fraction of retained hubs is sufficient to preserve navigability.

Moreover, while we restrict the number of outgoing connections when a node is first inserted into the graph (Line 8), we allow all nodes to form bidirectional links with newly inserted nodes up to the higher threshold M (Line 11), instead of the lower limit m. If a node exceeds this budget, we prune its neighbor list with the same RNG-style shrink heuristic rather than random deletion. This design ensures that each node retains the opportunity to connect with high-degree hub nodes, thereby preserving graph navigability with minimal impact on search quality.

6 Index Building and Update

Storage-Efficient Index Build. The naive index construction in LEANN requires precomputing embeddings for all objects to build the graph structure. Although these embeddings are discarded afterward to reduce storage at query time, the peak storage usage during construction can still be substantial. To address this, LEANN introduces a simple yet effective sharded merging pipeline strategy that builds the index efficiently under a user-specified storage constraint while preserving graph quality. The sharded merging

pipeline process consists of three stages: ① *Soft assignment with k-means.* We first run *k-means* on a small subset of the corpus to obtain k centroids. Each object is then embedded and assigned to its two nearest centroids. This is performed sequentially; after assignment, embeddings are immediately discarded, and only the two-centroid mapping for each passage is retained. ② *Shard-wise graph construction.* After the assignment, we build the graph index separately for each of the k shards. For each shard, embeddings are recomputed, the graph is constructed, and the embeddings are discarded. Since each passage belongs to two shards, the merged graph achieves good global connectivity. In total, each passage is embedded once for soft assignment and up to twice more during shard construction. This extra build-time compute is the price of reducing peak storage usage during construction, and can be mitigated in several ways: using cheaper assignment policies such as random assignment, keeping a transient embedding cache across nearby shards, or ordering shards to maximize cache reuse. ③ *Graph merging.* We then merge the k shard graphs into a single structure. For nodes appearing in two shards, we assign the higher of their two HNSW levels as the final layer. For lower layers, we merge their edge lists and randomly drop edges when the node degree exceeds M. This heuristic yields a well-connected, high-quality graph (see Figure 8); more advanced merging strategies, such as RNG-based pruning (Jaromczyk & Toussaint, 1992), are left for future work.

Efficient Index Update. We enable efficient updates in LEANN through a series of optimizations that substantially reduce computational and storage overhead. For a single update request, the naive recomputation procedure has a complexity of $\mathcal{O}(M \cdot efC + efC^2 + M^3)$, arising from repeated embedding calculations and neighbor maintenance. These three terms correspond to neighbor search, neighbor selection, and reverse-edge selection and updates. LEANN improves efficiency through lightweight embedding, caching and a simplified selection strategy, eliminating redundant computations and reducing the total cost to $\mathcal{O}(M \cdot efC)$ while preserving graph connectivity and quality. For deletions, LEANN employs soft deletion by marking nodes as inactive rather than removing them from the graph structure, preserving connectivity while avoiding costly graph reorganization. Details are provided in Appendix B.

Beyond single-node insertion, LEANN supports *batched add* operations with system-level optimizations. Incoming embeddings are temporarily buffered, and upon receiving a query, LEANN scans the buffer and merges its results with those from the existing graph. The buffered entries are then inserted asynchronously, amortizing update costs. This design minimizes computation and peak storage usage while maintaining low search latency.

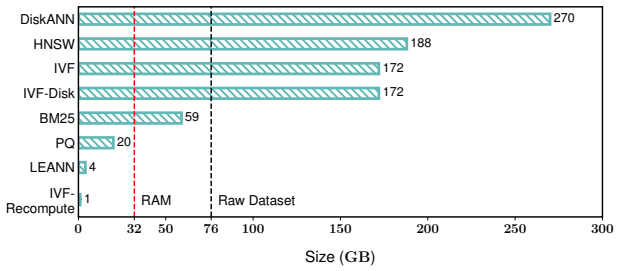


Figure 3. Storage consumption of different vector index methods on the RPJ-Wiki dataset. The black dashed line marks the raw dataset size (76 GB), while the red dashed line shows the typical RAM capacity (32 GB, RTX 4090, following our testbed configuration in §7.1). Memory-heavy methods like HNSW exceed this RAM limit and cannot run on such hardware. LEANN achieves the lowest storage footprint at only 5% of the original dataset size.

7 EVALUATION

We begin by describing the experimental setup in §7.1. Then, in §7.2, we present the main results and answer the following key questions: (1) What is the storage overhead of different indexing methods? (2) What is the latency of various vector search methods and the end-to-end RAG pipeline using them? (3) What is the end-to-end RAG accuracy achieved by different methods? Finally, in §7.3, we conduct comprehensive ablation studies to evaluate the effectiveness of each component in LEANN.

7.1 Experiment Settings

Workloads: Datastore and QA dataset. We construct the retrieval datastore using the RPJ-Wiki dataset (Computer, 2023), a widely used corpus comprising approximately **76 GB** of raw Wikipedia text. Following prior work (Shao et al., 2024), we segment the text into 256-token chunks and generate an embedding for each chunk using CONTRIEVER (Izacard et al., 2021), yielding 768-dimensional vectors. In total, we obtain 60 million ($N = 60$M) passages, producing about 173 GB of embeddings. For the QA datasets, we adopt four standard benchmarks commonly used in RAG and open-domain retrieval: NQ (Kwiatkowski et al., 2019), TriviaQA (Joshi et al., 2017), GPQA (Rein et al., 2024), and HotpotQA (Yang et al., 2018). Beyond the Wikipedia QA task, we further evaluate on FinanceBench (Islam et al., 2023) for *financial document retrieval*, the Enron Email Corpus (Ryan et al., 2024) for *email retrieval*, and LAION (Schuhmann et al., 2021) for *image data retrieval*.

Testbed. We evaluate our system on two hardware platforms. The first is a workstation with an NVIDIA RTX 4090 GPU (NVIDIA, 2022), 32GB RAM, and a 1 TB disk running WSL2. The second is an AWS EC2 M1 Mac instance (AWS, 2023) with an Apple M1 Ultra (Arm64) pro-

Table 2. Vector search and end-to-end RAG latency across datasets on RTX 4090. Latency is reported at 90% recall. Results for PQ and BM25 are omitted as they fail to reach this accuracy (their downstream accuracy is also low in Figure 4). Results for HNSW and IVF are measured on a server with larger memory, as they cause OOM on the local RTX 4090. Overhead (%) denotes the ratio of retrieval latency to total pipeline latency (retrieval / [retrieval + generation]).

Dataset	Generation (s)	Method	Retrieval (s)	Overhead (%)	Dataset	Generation (s)	Method	Retrieval (s)	Overhead (%)
NQ	20.86	HNSW	0.05	0.20	GPQA	69.60	HNSW	0.04	0.06
		IVF	2.55	10.90			IVF	0.17	0.25
		DiskANN	0.03	0.10			DiskANN	0.03	0.05
		IVF-Disk	3.44	14.10			IVF-Disk	0.06	0.09
		IVF-Recompute	307.61	93.60			IVF-Recompute	21.88	23.20
		PQ Compression	–	–			PQ Compression	–	–
		BM25	–	–			BM25	–	–
		LEANN	**2.48**	**10.60**			**LEANN**	**1.12**	**1.60**
TriviaQA	17.17	HNSW	0.04	0.20	HotpotQA	23.28	HNSW	0.05	0.20
		IVF	3.54	17.10			IVF	3.87	14.20
		DiskANN	0.06	0.30			DiskANN	0.11	0.50
		IVF-Disk	3.65	17.50			IVF-Disk	5.05	17.80
		IVF-Recompute	399.12	95.90			IVF-Recompute	429.46	94.80
		PQ Compression	–	–			PQ Compression	–	–
		BM25	–	–			BM25	–	–
		LEANN	**2.96**	**14.70**			**LEANN**	**7.12**	**23.40**

Table 3. Storage usage and retrieval latency overhead of LEANN on personal datasets (RTX 4090). Overhead (%) follows the definition in Table 2, and Storage Savings (%) denote the storage consumption of LEANN relative to HNSW.

Dataset	Generation (s)	Retrieval (s)	Overhead (%)	Storage Savings (%)
FinanceBench	46.0	1.5	3	97
Enron	22.3	1.9	8	98
LAION	6.6	1.6	20	97

cessor, macOS, 128GB RAM and a 512 GB EBS volume.

Baselines. We compare LEANN against the following baselines: **HNSW** (Malkov & Yashunin, 2018), **IVF**, **DiskANN** (Subramanya et al., 2019), **IVF-Disk**, **IVF-Recompute** (Seemakhupt et al., 2024), **PQ Compression** (Jégou et al., 2011), and **BM25** (Craswell et al., 2021). These baselines cover graph-based, cluster-based, quantization-based, and lexical retrieval paradigms. Detailed configurations are provided in Appendix C.1.

7.2 Main results

Storage consumption. We compare the storage consumption of all baselines and LEANN in Figure 3. Among all methods, only LEANN and IVF-Recompute maintain total storage overhead below 5% of the raw dataset size (76 GB).

Most existing systems incur substantial overhead, up to 2.5× the raw data size, making them impractical for deployment on personal devices. HNSW stores every dense embedding along with its graph connections, where each node contains a 768-dimensional embedding vector and padding for up to 60 neighbors (the maximum degree). DiskANN further amplifies this overhead due to its sector-aligned layout: each node's embedding (768×4 bytes) and neighbor list (60×4 bytes) are padded to 4KB SSD sectors. It also requires an additional 30GB for PQ embeddings, yielding the largest footprint (270 GB) among all methods. IVF and IVF-Disk exhibit similar overheads, both dominated by the cost of storing full embeddings. For BM25, the index size scales with the vocabulary and is roughly comparable to the raw corpus size in our setup. PQ compresses embeddings to a similar size as LEANN (5GB) but requires an additional 15GB for graph index metadata. In contrast, LEANN stores only a compact graph and highly compressed PQ embeddings, resulting in less than 5% additional storage overhead. Among all baselines, IVF-Recompute achieves the smallest footprint by storing only IVF centroids on disk.

We also compare LEANN against the most widely used HNSW index in Table 3, showing that it achieves over 97% storage savings across diverse datasets.

Latency evaluation for vector search and RAG. We evaluate the latency of vector search and end-to-end RAG across different methods in Table 2 and Table 3. We measure the per-request retrieval latency required to achieve 90% recall (Recall@3, defined in §2) and the subsequent generation time. Detailed measurement procedures are provided in Appendices C.2–C.3.

First, we show that ***second-level retrieval latency is acceptable***. The generation phase dominates the total response time, typically exceeding 10s and reaching up to 70s, so the

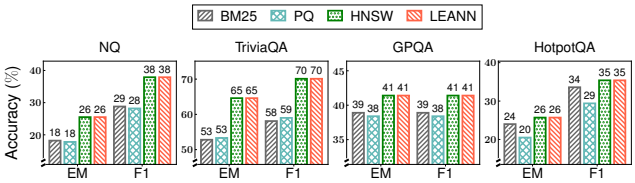


Figure 4. Comparison of Exact Match and F1 scores for downstream RAG tasks across four methods: keyword search (BM25), PQ-compressed vector search, HNSW, and LEANN. HNSW and LEANN are configured to achieve a target recall of 90%, while the PQ baseline is given extended search time to reach its highest possible recall. Here we use *Qwen3-4B* as the generation model.

design choice in LEANN to trade a small amount of latency for substantial storage savings is well justified.

Second, ***while several methods achieve storage efficiency, only LEANN delivers both high speed and accuracy.*** Among all baselines, only BM25, PQ, IVF-Recompute, and LEANN maintain storage overhead below the size of the raw dataset, as shown in Figure 3. However, BM25 and PQ exhibit low retrieval accuracy and fail to reach 90% recall. IVF-Recompute attains high recall but requires up to two orders of magnitude longer retrieval time than LEANN (up to 200× slower).

This difference arises because LEANN employs a graph-based index with $\mathcal{O}(\log N)$ embedding recomputation, while IVF-Recompute performs $\mathcal{O}(\sqrt{N})$ recomputations (Wang et al., 2021) (N=60M in our experiments). Additional latency optimizations described in §4 further contribute to LEANN's performance advantage.

Finally, although LEANN's standalone vector search latency is higher than graph-based baselines such as HNSW and DiskANN, we show that ***LEANN introduces negligible latency overhead when integrated into the full RAG pipeline*** on personal devices, while using far less storage. As shown in Table 2 and Table 3, LEANN consistently adds less than 20% latency overhead to the end-to-end retrieval and generation process. For reasoning-intensive tasks such as the graduate-level QA benchmark GPQA, the additional overhead introduced by LEANN remains under 3%, as the model's long chain-of-thought generation dominates total latency.

We include Mac latency results in Table 4 (Appendix C.4), using the same setup as Table 2. Despite the Mac's lower TFLOPS, all previous conclusions hold, demonstrating LEANN's generalization across platforms.

Accuracy of downstream RAG applications. To evaluate RAG accuracy, we compare all retrieval methods on four QA datasets using Exact Match (EM) and F1 as metrics, as shown in Figure 4. LEANN achieves the highest downstream QA performance among all methods.

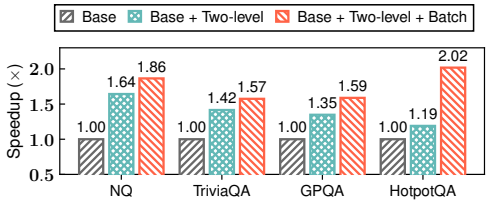


Figure 5. Speedup achieved by different optimization techniques described in §4 when evaluated on four datasets to reach the same recall level. *Two-level* refers to the optimization in §4.1, while *Batch* corresponds to §4.2.

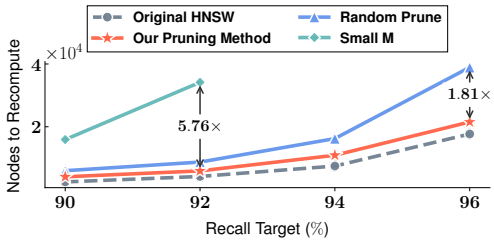


Figure 6. Comparison of pruned graph quality against two heuristic methods using the datastore in §7.1. At each recall target on the NQ dataset, we vary the search queue length ef to determine the minimum number of nodes that must be recomputed (lower is better). The dashed gray line represents the original HNSW graph as a baseline reference, which uses twice the storage (i.e., average degree) of the pruned methods.

As shown in Figure 4, LEANN consistently outperforms BM25 and PQ across all datasets. It improves EM by up to 11.8% over BM25 and 11.3% over PQ, and F1 by up to 12.0% and 11.1%, respectively. The gains are most pronounced on factual answering benchmarks such as NQ and TriviaQA, where accurate semantic retrieval provides clear benefits. In contrast, the improvement is smaller on GPQA and HotpotQA. This is because RPJ-Wiki datastore is partially out-of-distribution for GPQA, which contains graduate-level questions that are less supported by Wikipedia content, and HotpotQA requires multi-hop reasoning, while our setup performs only single-hop retrieval.

Finally, when a target recall level (i.e., 90%) is enforced, the downstream accuracy of LEANN matches that of HNSW, confirming that our method preserves accuracy while achieving substantial storage savings.

7.3 Ablation Studies and Micro Benchmarks

Latency optimization techniques. To evaluate the latency optimizations in LEANN described in §4, we incrementally enable each component while maintaining a fixed target recall across multiple datasets. Starting from a naive HNSW recomputation baseline, adding the two-level search mechanism (§4.1) yields an average 1.4× speedup (up to 1.6×) by

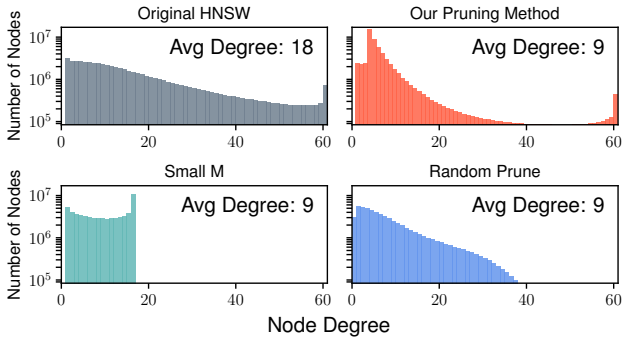


Figure 7. Comparison of (out-)degree distributions among the original graph, our pruning method, and two heuristic baselines. Similar to Figure 6, the gray curve represents the original HNSW graph, which has twice the size of the others. Only our pruning method successfully preserves the high-degree nodes.

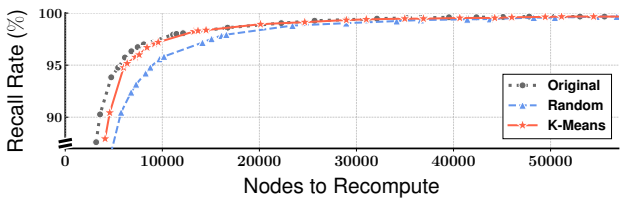


Figure 8. Comparison of storage-efficient index construction methods with the original HNSW.

reducing the number of nodes requiring recomputation, with LEANN enabling lightweight distance estimation without invoking the embedding generator. Incorporating dynamic batching further improves GPU utilization during recomputation, increasing the average speedup to $1.8\times$ and the peak to $2.0\times$. Among all datasets, HotpotQA gains the most from dynamic batching, as its longer search paths allow more effective grouping of multi-hop requests.

Alternative graph pruning methods. We compare our high-degree preserving graph pruning algorithm with two baselines: *(1) Random Prune*, which randomly removes 50% of edges from the original graph; and *(2) Small M*, which constrains the maximum out-degree during graph construction, yielding an average degree half that of the original graph. We evaluate graph quality by measuring the number of nodes that must be recomputed to achieve a given recall target, as shown in Figure 6. In LEANN, latency is dominated by embedding recomputations, making this metric a proxy for retrieval latency. The original graph has an average degree of 18. All three pruning methods, ours and the two baselines, are applied to reduce the average degree by half, from degree of 18 to 9, thereby halving the graph's storage overhead. As shown in Figure 6, our pruning method introduced in §5 achieves performance comparable to the original unpruned graph, while using only half the edges. To reach the same recall levels, Random Prune requires up to $1.8\times$ more nodes to recompute, while Small M requires up to $5.8\times$ more nodes to recompute. We omit the Small M results at the 94% and 96% recall targets, as it fails to reach these accuracy levels.

Degree distribution in pruned graphs. To better understand the effectiveness of our pruning strategy, we analyze the out-degree distributions of the original graph, our approach, Random Prune, and Small M. As discussed in §5, our design explicitly aims to preserve high-degree "hub" nodes. As shown in Figure 7, it successfully retains a substantial number of such nodes, whereas the other two baselines fail to do so. This underscores the critical role of hub nodes in supporting efficient graph-based vector search, a finding that aligns with insights from prior work (Ren et al., 2020; Munyampirwa et al., 2024; Manohar et al., 2024).

Storage-efficient index construction. We evaluate the storage-efficient index construction techniques introduced in §6, comparing them against the standard HNSW construction. Specifically, we implement two variants of the proposed sharded merging pipeline approach: (1) a *k-means*–based method (see §6) that groups similar passages before sharding, and (2) a random assignment baseline that omits clustering. We assess graph quality using the same methodology as before, with results shown in Figure 8. In this setup, the dataset is partitioned into 15 shards, achieving about a **5**$\times$ reduction in peak storage usage during index construction.

The *k-means*–sharded graph achieves nearly the same recall as the original HNSW with only a small increase in recomputation cost, indicating that it maintains strong connectivity after sharding and merging. In contrast, the randomly sharded graph requires much more recomputation to reach the same recall. These results highlight the benefit of clustering similar passages before sharding, validating the design of our sharded merging pipeline approach.

Index update. Figure 9 shows how latency changes as we incrementally enable the optimizations of the LEANN add operation described in Appendix B. We achieve up to a $63.3\times$ speedup over the naive method, consistent with our theoretical analysis. On the right, introducing a buffer to delay batched additions further improves search speed while maintaining accuracy.

More experiments. We provide additional experiment results in Appendix D. In Appendix D.1, we show that using a smaller embedding model further accelerates LEANN without compromising accuracy, and in Appendix D.2 we decompose the query latency of LEANN into PQ lookup, text processing, and embedding recomputation.

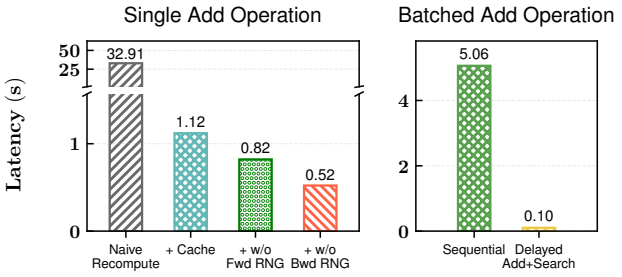


Figure 9. Comparison of update methods

8 Related Work

Resource-constrained vector search. Many works aim to reduce the memory cost of vector search. Disk-based systems like DiskANN (Subramanya et al., 2019) store vectors and graphs on disk with compressed in-memory embeddings for navigation. Starling (Wang et al., 2024) improves disk I/O, and FusionANNS (Tian et al., 2025) coordinates SSD, CPU, and GPU to lower cost. AiSAQ (Tatsuno et al., 2024) and LM-DiskANN (Pan et al., 2023) further cut DRAM use by keeping compressed embeddings on disk. EdgeRAG (Seemakhupt et al., 2024) generates embeddings online via an IVF-based index but still suffers high storage and recomputation overhead. MicroNN (Zhang et al., 2025) and ObjectBox (ObjectBox, 2025) are optimized for personal devices but still require storing all embeddings. Embedding compression methods like PQ (Jégou et al., 2011) and RabitQ (Gao & Long, 2024) save space but lose accuracy under tight budgets. In contrast, LEANN combines on-the-fly embedding recomputation with a pruned graph index and optimized traversal for personal devices.

Vector search applications on personal devices. On-device vector search enables privacy-preserving, low-latency, and offline capabilities across diverse applications. On-device RAG systems ground language models in personal document collections while maintaining data privacy (Ryan et al., 2024; Wang & Chau, 2024; Lee et al., 2024; Zerhoudi & Granitzer, 2024). Personalized recommenders (Yin et al., 2024) match user profiles with item embeddings on the device, while vision-based search (Ren et al., 2023) retrieves local images or videos to assist downstream QA or generation tasks. These applications motivate the design of LEANN to enable efficient, low-overhead vector search on personal devices.

9 Discussion

Storage–compute trade-off. LEANN is not restricted to a single operating point. When disk constraints are relaxed, it can materialize a small set of frequently accessed embeddings to reduce recomputation. As shown in Figure 10, caching just 10% of the original embeddings yields up to a $1.5\times$ speedup with a cache-hit rate of up to 41.9%, providing a smooth storage–latency trade-off curve.

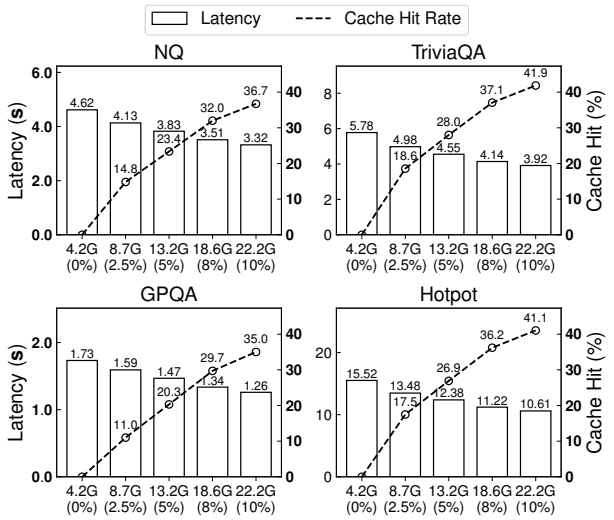


Figure 10. Latency and cache-hit rate under relaxed storage budgets. Materializing a small fraction of exact embeddings exposes a smooth storage–latency trade-off for LEANN.

Intended scope and limitations. LEANN is designed for storage-constrained environments where modest recomputation cost can be amortized across the overall workload or where access frequency is low. Representative deployment scenarios include:

- *LLM-centric pipelines* such as RAG, agent memory, and search-augmented agents, where generation dominates end-to-end latency.
- *Low-QPS local and edge-device workloads* such as personal search, ad-hoc log analysis, and on-device document retrieval, where query rates are modest and storage is the binding constraint.
- *Compute-rich datacenter settings*, where recomputation can be overlapped with other work and storage remains costly at scale.

Under these workload characteristics, LEANN's recomputation-based design offers a favorable storage–latency trade-off, consistent with the five-minute rule (Gray & Graefe, 1997).

For workloads with shorter generation times, the retrieval overhead becomes more visible. One mitigation is to pipeline retrieval with generation, as in PipeRAG (Jiang et al., 2024), feeding partial results to the LLM early to hide retrieval latency; this is orthogonal to LEANN. Pure retrieval-only, high-throughput workloads with strict tail-latency requirements (e.g., real-time content search, recom-

mendation systems) are not the primary target for LEANN.

Generality and future work. The core techniques of LEANN, including on-the-fly recomputation, hub-preserving graph pruning, and recomputation-aware search, are broadly applicable across graph-based ANN frameworks. Furthermore, LEANN does not fundamentally require neural accelerators. Lightweight, lookup-based embedding models such as Model2Vec (Tulkens & van Dongen, 2024) compute embeddings by aggregating token-level vectors from a static pretrained table, achieving near memory-lookup-level compute cost. Pairing such models with LEANN can substantially lower recomputation cost on CPU-only devices while retaining the storage benefits, extending recomputation-based indexing to accelerator-free deployments.

10 Conclusions

Similarity search over high-dimensional embeddings underpins many generative AI applications such as RAG. However, enabling such capabilities remains challenging due to the substantial storage required for embeddings and index metadata. We present LEANN, a storage-efficient vector index based on *graph-based recomputation*. By combining *two-level search* with *dynamic batching*, LEANN supports efficient query processing without storing the full embedding set. A *high-degree preserving pruning* strategy further reduces graph storage while maintaining accuracy. LEANN also offers fast, storage-efficient index construction and update pipelines. Together, these techniques allow LEANN to operate with an index smaller than 5% of the raw data size, achieving up to 50× storage reduction compared to existing methods while preserving high recall and low latency.

References

Asai, A., Wu, Z., Wang, Y., Sil, A., and Hajishirzi, H. Self-rag: Learning to retrieve, generate, and critique through self-reflection. In *The Twelfth International Conference on Learning Representations*, 2023.

Aumüller, M., Bernhardsson, E., and Faithfull, A. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems*, 87:101374, 2020.

AWS. Amazon EC2 G5 instance. https://aws.amazon.com/ec2/instance-types/mac/, 2023. [Online; accessed April-2025].

Bai, S., Chen, K., Liu, X., Wang, J., Ge, W., Song, S., Dang, K., Wang, P., Wang, S., Tang, J., et al. Qwen2. 5-vl technical report. *arXiv preprint arXiv:2502.13923*, 2025.

Cai, D., Wang, S., Peng, C., et al. Recall: Empowering multimodal embedding for edge devices. arXiv:2409.15342, 2024.

Computer, T. RedPajama: An open source recipe to reproduce LLaMA training dataset. https://github.com/togethercomputer/RedPajama-Data, 2023. Accessed: May 10, 2025.

Craswell, N., Mitra, B., Yilmaz, E., Campos, D., and Lin, J. Ms marco: Benchmarking ranking models in the large-data regime. In *proceedings of the 44th International ACM SIGIR conference on research and development in information retrieval*, pp. 1566–1576, 2021.

Douze, M., Sablayrolles, A., and Jégou, H. Link and code: Fast indexing with graphs and compact regression codes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3646–3654, 2018.

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

Fu, C., Xiang, C., Wang, C., and Cai, D. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proc. VLDB Endow.*, 12(5):461–474, January 2019. ISSN 2150-8097. doi: 10.14778/3303753.3303754. URL https://doi.org/10.14778/3303753.3303754.

Gao, J. and Long, C. RabitQ: Quantizing high-dimensional vectors with a theoretical error bound for approximate nearest neighbor search. In *Proceedings of the ACM on Management of Data (SIGMOD '24)*, volume 2, 2024.

Gray, J. and Graefe, G. The five-minute rule ten years later, and other computer storage rules of thumb. *ArXiv*, cs.DB/9809005, 1997. URL https://api.semanticscholar.org/CorpusID:21524661.

He, Y., Sainath, T. N., Prabhavalkar, R., McGraw, I., Alvarez, R., Zhao, D., et al. Streaming end-to-end speech recognition for mobile devices. In *Proc. IEEE ICASSP*, pp. 6381–6385, 2019.

Henzinger, A., Dauterman, E., Corrigan-Gibbs, H., and Zeldovich, N. Private web search with tiptoe. Cryptology ePrint Archive, Paper 2023/1438, 2023. URL https://eprint.iacr.org/2023/1438.

Islam, P., Kannappan, A., Kiela, D., Qian, R., Scherrer, N., and Vidgen, B. Financebench: A new benchmark for financial question answering, 2023. URL https://arxiv.org/abs/2311.11944.

Izacard, G., Caron, M., Hosseini, L., Riedel, S., Bojanowski, P., Joulin, A., and Grave, E. Unsupervised dense information retrieval with contrastive learning. *arXiv preprint arXiv:2112.09118*, 2021.

Jaromczyk, J. and Toussaint, G. Relative neighborhood graphs and their relatives. *Proceedings of the IEEE*, 80 (9):1502–1517, 1992. doi: 10.1109/5.163414.

Jiang, W., Zhang, S., Han, B., Wang, J., Wang, B., and Kraska, T. Piperag: Fast retrieval-augmented generation via algorithm-system co-design. *arXiv preprint arXiv:2403.05676*, 2024.

Joshi, M., Choi, E., Weld, D. S., and Zettlemoyer, L. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. *arXiv preprint arXiv:1705.03551*, 2017.

Jégou, H., Douze, M., and Schmid, C. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2011. doi: 10.1109/TPAMI.2010.57.

Kwiatkowski, T., Palomaki, J., Redfield, O., Collins, M., Parikh, A., Alberti, C., Epstein, D., Polosukhin, I., Devlin, J., Lee, K., Toutanova, K., Jones, L., Kelcey, M., Chang, M.-W., Dai, A. M., Uszkoreit, J., Le, Q., and Petrov, S. 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/.

Lee, C., Prahlad, D., Kim, D., and Kim, H. Work-in-progress: On-device retrieval augmented generation with knowledge graphs for personalized large language models. In *2024 International Conference on Embedded Software (EMSOFT)*, pp. 1–1, 2024. doi: 10.1109/EMSOFT60242.2024.00006.

Lempitsky, V. The inverted multi-index. In *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, CVPR '12, pp. 3069–3076, USA, 2012. IEEE Computer Society. ISBN 9781467312264.

Li, Z., Zhang, X., Zhang, Y., Long, D., Xie, P., and Zhang, M. Towards general text embeddings with multi-stage contrastive learning. *arXiv preprint arXiv:2308.03281*, 2023.

Lin, J., Ma, X., Lin, S.-C., Yang, J.-H., Pradeep, R., and Nogueira, R. Pyserini: A Python toolkit for reproducible information retrieval research with sparse and dense representations. In *Proceedings of the 44th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2021)*, pp. 2356–2362, 2021.

Mageirakos, V., Wu, B., and Alonso, G. Cracking vector search indexes. *arXiv preprint arXiv:2503.01823*, 2025.

Malkov, Y. A. and Yashunin, D. A. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.

Manohar, M. D., Shen, Z., Blelloch, G., Dhulipala, L., Gu, Y., Simhadri, H. V., and Sun, Y. 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*, pp. 270–285, 2024.

Mohoney, J., Pacaci, A., Chowdhury, S. R., Mousavi, A., Ilyas, I. F., Minhas, U. F., Pound, J., and Rekatsinas, T. High-throughput vector similarity search in knowledge graphs. *Proceedings of the ACM on Management of Data*, 1(2):1–25, 2023.

Munyampirwa, B., Lakshman, V., and Coleman, B. Down with the hierarchy: The'h'in hnsw stands for" hubs". *arXiv preprint arXiv:2412.01940*, 2024.

NVIDIA. Nvidia geforce rtx 4090 graphics card. https://www.nvidia.com/en-us/geforce/graphics-cards/40-series/rtx-4090/, 2022. Accessed: 2025-08-20.

ObjectBox. On-device vector search. https://docs.objectbox.io/on-device-vector-search, 2025. Accessed: May 15, 2025.

Pan, Y., Sun, J., and Yu, H. Lm-diskann: Low memory footprint in disk-native dynamic graph-based ann indexing. In *2023 IEEE International Conference on Big Data (BigData)*, pp. 5987–5996, 2023. doi: 10.1109/BigData59044.2023.10386517.

Rein, D., Hou, B. L., Stickland, A. C., Petty, J., Pang, R. Y., Dirani, J., Michael, J., and Bowman, S. R. Gpqa: A graduate-level google-proof q&a benchmark. In *First Conference on Language Modeling*, 2024.

Rekabsaz, N., Lesota, O., Schedl, M., Brassey, J., and Eickhoff, C. Tripclick: the log files of a large health web search engine. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 2507–2513, 2021.

Ren, J., Zhang, M., and Li, D. Hm-ann: efficient billion-point nearest neighbor search on heterogeneous memory. In *Proceedings of the 34th International Conference on*

Neural Information Processing Systems, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc. ISBN 9781713829546.

Ren, J., Tulyakov, S., Peng, K.-C., Wang, Z., and Shi, H. Efficient neural networks: From algorithm design to practical mobile deployments. CVPR 2023 Tutorial, 2023. https://snap-research.github.io/efficient-nn-tutorial/.

Research, F. A. Guidelines to choose an index. https://github.com/facebookresearch/faiss/wiki/Guidelines-to-choose-an-index/28074dc0ddc733f84b06fa4d99b3f6e2ef65613d#if-below-1m-vectors-ivfx, 2025. Accessed: 2025-05-10.

Ryan, M. J., Xu, D., Nivera, C., and Campos, D. EnronQA: Towards personalized RAG over private documents. *arXiv preprint arXiv:2505.00263*, 2024.

Schuhmann, C., Vencu, R., Beaumont, R., Kaczmarczyk, R., Mullis, C., Katta, A., Coombes, T., Jitsev, J., and Komatsuzaki, A. Laion-400m: Open dataset of clip-filtered 400 million image-text pairs. *arXiv preprint arXiv:2111.02114*, 2021.

Seemakhupt, K., Liu, S., and Khan, S. Edgerag: Online-indexed rag for edge devices. *arXiv preprint arXiv:2412.21023*, 2024.

Severo, D., Ottaviano, G., Muckley, M., Ullrich, K., and Douze, M. Lossless compression of vector ids for approximate nearest neighbor search. *arXiv preprint arXiv:2501.10479*, 2025.

Shao, R., He, J., Asai, A., Shi, W., Dettmers, T., Min, S., Zettlemoyer, L., and Koh, P. W. W. Scaling retrieval-based language models with a trillion-token datastore. *Advances in Neural Information Processing Systems*, 37: 91260–91299, 2024.

Shen, M., Umar, M., Maeng, K., Suh, G. E., and Gupta, U. Towards understanding systems trade-offs in retrieval-augmented generation model inference, 2024. URL https://arxiv.org/abs/2412.11854.

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

Tatsuno, K., Miyashita, D., Ikeda, T., Ishiyama, K., Sumiyoshi, K., and Deguchi, J. AiSAQ: all-in-storage ANNS with product quantization for DRAM-free information retrieval. *arXiv preprint arXiv:2404.06004*, 2024. URL https://arxiv.org/abs/2404.06004.

Tian, B., Liu, H., Tang, Y., Xiao, S., Duan, Z., Liao, X., Jin, H., Zhang, X., Zhu, J., and Zhang, Y. Towards high-throughput and low-latency billion-scale vector search via CPU/GPU collaborative filtering and re-ranking. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pp. 171–185, Santa Clara, CA, February 2025. USENIX Association. ISBN 978-1-939133-45-8. URL https://www.usenix.org/conference/fast25/presentation/tian-bing.

Toussaint, G. T. The relative neighbourhood graph of a finite planar set. *Pattern Recognit.*, 12:261–268, 1980. URL https://api.semanticscholar.org/CorpusID:2830642.

Tulkens, S. and van Dongen, T. Model2vec: Fast state-of-the-art static embeddings, 2024. URL https://github.com/MinishLab/model2vec.

Wang, M., Xu, X., Yue, Q., and Wang, Y. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *arXiv preprint arXiv:2101.12631*, 2021.

Wang, M., Xu, W., Yi, X., Wu, S., Peng, Z., Ke, X., Gao, Y., Xu, X., Guo, R., and Xie, C. Starling: An i/o-efficient disk-resident graph index framework for high-dimensional vector similarity search on data segment. In *Proceedings of the ACM on Management of Data (SIGMOD '24)*, volume 2, 2024. doi: 10.1145/3639269.3652200.

Wang, Z. J. and Chau, D. H. Mememo: On-device retrieval augmentation for private and personalized text generation. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 2765–2770, 2024.

Yang, A., Li, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Gao, C., Huang, C., Lv, C., et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Yang, Z., Qi, P., Zhang, S., Bengio, Y., Cohen, W. W., Salakhutdinov, R., and Manning, C. D. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. *arXiv preprint arXiv:1809.09600*, 2018.

Yin, H., Chen, T., Qu, L., and Cui, B. On-device recommender systems: A comprehensive survey. *arXiv preprint arXiv:2401.11441*, 2024.

Zerhoudi, S. and Granitzer, M. Personarag: Enhancing retrieval-augmented generation systems with user-centric agents. *arXiv preprint arXiv:2407.09394*, 2024.

Zhang, Z., Li, Z., Zhang, Z., and Yu, P. S. MicroNN: An embedded vector search engine for low-resource environments. *arXiv preprint arXiv:2504.05573*, 2025.

Zhu, J., Patel, L., Zaharia, M., and Popa, R. A. Compass: Encrypted semantic search with high accuracy. Cryptology ePrint Archive, Paper 2024/1255, 2024. URL https://eprint.iacr.org/2024/1255.

A RNG PRUNING

For a node v inserted into any proximity-graph index (including HNSW), the algorithm first searches for a list of candidate neighbors a, b, c, d (see Algorithm 1) sorted by distance from v. The RNG-based pruning rule(Jaromczyk & Toussaint, 1992; Toussaint, 1980), implemented in Line 6, then iterates through this list in order of increasing distance. A candidate x is pruned if there exists a closer neighbor a such that $\text{Dist}(a, x) < \text{Dist}(v, x)$. This effectively removes the longest edge in the triangle formed by (v, a, x). As illustrated in Figure 11, the edges $v - b$ and $v - c$ are pruned, so the search from v to b or c proceeds indirectly through a. This pruning strategy is widely used in modern graph-based ANN indexes and makes the resulting graph extremely sparse (Munyampirwa et al., 2024; Malkov & Yashunin, 2018).

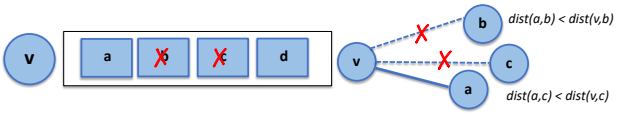


Figure 11. Select neighbors from candidate nodes using RNG.

B LEANN UPDATE STRATEGY

The ADD algorithm is presented in Algorithm 4.

B.1 Add Operation: Method and Time Complexity

Naive Implementation. A naive implementation of the ADD operation in LEANN recomputes all distances from scratch since only the graph structure is stored. Its total time complexity can be expressed as:

$$O(M \cdot efC + efC^2 + M^3),$$

where efC is the construction queue length and M the maximum node degree.

We first analyze the complexity of SHRINKNEIGHBORLIST. Each node placed in the retained set R may be re-examined up to $|W|$ times, giving a cost of $O(|W|^2)$ for a single call.

Specifically:

- SEARCHNEIGHBORSTOADD performs a one-time neighbor search without revisiting nodes, yielding $O(M \cdot efC)$. Caching offers no benefit since nodes are not revisited.
- SHRINKNEIGHBORLIST (Line 12) runs in $O(efC^2)$, as it computes pairwise distances among up to efC candidates.
- Adding forward edges requires no recomputation, since each node maintains at most M links.
- Adding reverse edges costs $O(M^3)$, as up to M neighbors are updated and each triggers an $O(M^2)$ RNG-based shrink.

Caching Optimization. To improve efficiency, LEANN introduces a distance cache to eliminate redundant computations in the SHRINK step, reducing the overall complexity to:

$$O(M \cdot efC + efC + M^2),$$

since the shrink operation now costs only $O(|W|)$.

Simplified RNG Pruning. By further simplifying SHRINKNEIGHBORLIST to randomly select neighbors instead of performing full RNG checks, the complexity becomes:

$$O(M \cdot efC + M^2).$$

Finally, applying the same simplification to the reverse-edge update step yields the optimized complexity:

$$O(M \cdot efC),$$

reducing the cost from cubic to linear in M while maintaining comparable graph connectivity.

B.2 Batched Add Operation: Optimization

When a batch of add operations is followed by a search request, LEANN does not immediately insert all new passages. Instead, it temporarily buffers their embeddings and merges search results from both the existing graph and the buffered embeddings. After the search completes, the buffered passages are inserted *asynchronously*, a process we term *delayed insertion*.

The same system optimizations described earlier can be reused here, with the addition of a global cache to avoid redundant computations across multiple add requests. To maintain storage efficiency, LEANN monitors the cache size and clears it once a predefined budget is reached, starting a new round of batched insertion.

B.3 Soft Deletion Strategy

LEANN adopts a simple soft delete for graph nodes. Each node keeps a binary delete flag, so removal is an $O(1)$ update that leaves the adjacency list untouched. During query processing, we still traverse deleted nodes to reach their neighbors, but before producing results we filter the candidate queue (the EQ in Algorithm 2) by this flag and then take the top-k active entries to guarantee correctness.

If the fraction of deleted nodes grows beyond a threshold (e.g., 5%), we can trigger a background rebuild. Exploring such policies is left for future work.

Algorithm 4 Node Insertion into Graph Index

Input: Existing graph G; construction queue length efC; maximum degree M; node to insert v
Output: Updated graph G including node v
function SHRINK(W, M)
    Initialize $R \leftarrow \emptyset$
    for x in W (by ascending distance to query q) **do**
        if no $y \in R$ s.t. $\text{Dist}(x, y) < \text{Dist}(x, q)$ **then**
            Add x to R
        if $|R| = M$ **then**
            break
    return R
$W \leftarrow$ SEARCH(v, efC) ▷ See Algorithm 1
$W \leftarrow$ SHRINK(W, M) ▷ ShrinkNeighborList
Add directed edges from v to all nodes in W
for each $w \in W$ **do**
    Add directed edge from w to v
    if $\deg(w) > M$ **then**
        SHRINK(w's neighbor list, M)

C EVALUATION DETAILS

C.1 Baseline Configurations

- **HNSW** (Malkov & Yashunin, 2018): We use the `faiss.IndexHNSWFlat` implementation with construction parameters recommended by FAISS: M=30 and $efConstruction$=128, distinct from the search-time parameter ef.

- **IVF** (Lempitsky, 2012): We adopt the `faiss.IndexIVFFlat` implementation. Following best practices from FAISS (Research, 2025) and prior work (Henzinger et al., 2023), we set the number of centroids to $\sqrt{N}$, where N is the size of the datastore. For our N=60M setup, this corresponds to $nlist$=8192.

- **DiskANN** (Subramanya et al., 2019): We use DiskANN with M=60 and $efConstruction$=128, following recommended settings (Subramanya et al., 2019). It stores only a PQ table in memory and loads full embeddings from disk on demand.

- **IVF-Disk**: Reduces memory usage by employing memory-mapped files (`mmap`) instead of loading the entire index into memory. Implemented using FAISS's `faiss.contrib.ondisk` module with the same parameters as IVF.

- **IVF-Recompute** (Seemakhupt et al., 2024): Inspired by Edge-RAG, this variant recomputes embeddings at query time instead of storing them, using the same construction parameters as IVF.

- **PQ Compression** (Jégou et al., 2011): Applies Product Quantization to compress stored embeddings while preserving the graph structure. For fair comparison, we compress the vectors to 5 GB—slightly larger than our system's 4 GB footprint. We use the PQ implementation from (Subramanya et al., 2019).

- **BM25** (Craswell et al., 2021; Rekabsaz et al., 2021): A classical lexical ranking method widely used in keyword-based retrieval systems. We employ the standard implementation from Pyserini (Lin et al., 2021).

C.2 Latency Measurement and Evaluation Protocol

To evaluate retrieval accuracy, we report Recall@k as defined in §2. In open-domain settings, ground-truth labels for retrieved passages are typically unavailable. Following standard practice (Jégou et al., 2011; Schuhmann et al., 2021; Zhu et al., 2024), we treat the results from exact search as a proxy for ground truth. In all experiments, we set k=3, consistent with prior work (Shao et al., 2024; Asai et al., 2023), and report Recall@3 as our retrival accuracy metric.

For latency evaluation, we measure the time required to achieve different target recall levels. Specifically, we perform a binary search to find the minimal search queue length ef (as defined in Algorithm 1) that reaches the desired recall. Using the resulting ef, we record the average retrieval latency over 20 random queries.

C.3 Latency Measurement in RAG Pipeline

We evaluate the latency of LEANN at the 90% recall level across all datasets. For text generation, we use *Qwen3-4B* (Yang et al., 2025), and for multimodal workloads, we use *Qwen2.5-VL-7B-Instruct* (Bai et al., 2025). Both the embedding and generation models are implemented using the Hugging Face framework.

C.4 RAG Latency on Mac Platform

To validate the generalizability of our results across different hardware platforms, we conducted additional experiments on Mac hardware. Table 4 presents the vector search and end-to-end RAG latency measurements on Mac, following the same experimental protocol as the RTX 4090 results shown in the main paper. The results demonstrate that LEANN maintains its efficiency advantages on Mac hardware, with retrieval overhead remaining low compared to other methods despite the different underlying architecture and computational characteristics.

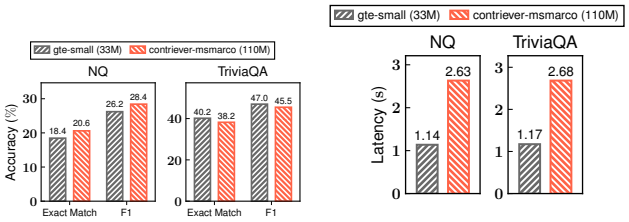
D MORE ABLATION STUDIES

D.1 Using Different Embedding Model Sizes

Since the primary bottleneck of our system lies in the recomputation process, we further explore the potential for latency

Table 4. Vector search and end-to-end RAG latency across datasets on Mac. Latency is reported at 90% recall. Results for PQ and BM25 are omitted as they fail to reach this accuracy (their downstream accuracy is also low in Figure 4). Results for HNSW and IVF are omitted, as they cause OOM on the local Mac and on all EC2 Mac instances on AWS. Overhead (%) denotes the ratio of retrieval latency to total pipeline latency (retrieval / [retrieval + generation]).

Dataset	Generation (s)	Method	Retrieval (s)	Overhead (%)	Dataset	Generation (s)	Method	Retrieval (s)	Overhead (%)
NQ	45.42	HNSW	–	–	GPQA	132.24	HNSW	–	–
		IVF	–	–			IVF	–	–
		DiskANN	0.37	0.8			DiskANN	0.29	0.2
		IVF-Disk	2.94	6.1			IVF-Disk	0.11	0.1
		IVF-Recompute	2446.60	98.2			IVF-Recompute	174.06	56.8
		PQ Compression	–	–			PQ Compression	–	–
		BM25	–	–			BM25	–	–
		LEANN	**13.84**	**23.4**			**LEANN**	**5.22**	**3.8**
TriviaQA	52.92	HNSW	–	–	HotpotQA	44.67	HNSW	–	–
		IVF	–	–			IVF	–	–
		DiskANN	0.98	1.8			DiskANN	1.91	4.1
		IVF-Disk	2.64	4.8			IVF-Disk	3.54	7.4
		IVF-Recompute	3174.41	98.4			IVF-Recompute	3415.74	98.7
		PQ Compression	–	–			PQ Compression	–	–
		BM25	–	–			BM25	–	–
		LEANN	**17.15**	**24.5**			**LEANN**	**44.80**	**50.1**



(a) Accuracy **(b)** Latency

Figure 12. **[Ablation Study]:** Downstream accuracy and end-to-end latency when swapping the embedding model for a lightweight alternative on a 2M-chunk datastore with `ef=50`.

reduction by adopting a smaller embedding model. Specifically, we replace the original *Contriever* model (110M parameters) used in §7.1 with the lightweight *GTE-small* model (Li et al., 2023), which has only 34M parameters. We evaluate performance on a 2M document datastore using a fixed search queue length of `ef=50`. As shown in Figure 12, *GTE-small* achieves a 2.3× speedup while maintaining downstream task accuracy within 2% of the Contriever baseline, demonstrating that LEANN can further reduce latency by leveraging lighter encoders without sacrificing answer quality.

D.2 Graph-based Recomputation Breakdown

Figure 13 decomposes the latency of a batched query into three stages: PQ lookup, text processing, and embedding recomputation. Each batch aggregates multiple hops of recomputation, as described in §4.2. First, LEANN performs PQ lookups to select promising nodes, then retrieves and tokenizes the corresponding raw text. The tokenized inputs

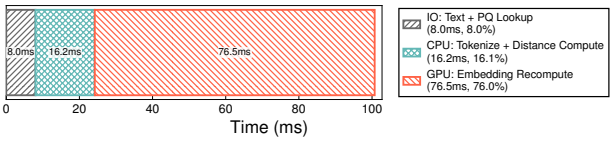


Figure 13. **[Ablation Study]:** Latency breakdown of a batch of requests during graph-based recomputation.

are sent to the local embedding generator. Finally, LEANN performs embedding recomputation and distance calculation. Although embedding recomputation is the primary bottleneck in LEANN, accounting for roughly 76% of total latency, the three stages span I/O, CPU, and GPU resources, indicating opportunities to overlap work and further improve efficiency.