Concurrent Updates and Queries on a Graph-Based Vector Index

15-618 Final Project Proposal

Hao Zhang, Hongyi Zhang

November 9, 2026

URL: https://samplealameda.github.io/concurrent-vector-index/

Summary

We will build an in-memory graph-based approximate nearest neighbor index from scratch in C++ that supports concurrent inserts, deletes, and searches on a single shared Vamana graph [9, 8], and study how these operations scale on multicore shared-memory CPUs. We will compare synchronization schemes of different granularity (a global reader–writer lock and per-node reader–writer locks, with optimistic lock-free reads as a stretch goal) and three published ways of repairing the graph after a delete, taken from FreshDiskANN [8], IP-DiskANN [10], and CleANN [11]. Every configuration will be evaluated on both search quality (recall@10 against brute-force ground truth) and performance (throughput and latency versus thread count and update ratio).

Background

Approximate nearest neighbor (ANN) search is the retrieval step in vector databases and retrieval-augmented language models. Documents are embedded as vectors with a few hundred dimensions, and a query asks for the k closest vectors. In practice, the indexed collection keeps changing while queries are being served, because documents are being added and removed. Asai et al. [2] list datastore updates as one of the main advantages of retrieval-augmented models, and point out efficient index updating is an open infrastructure problem. Graph-based indexes are the fastest option for static data [3, 7], but how to run inserts, deletes, and searches concurrently on one graph is still unsettled.

The index is a directed graph with one node per vector. Each node stores its vector and a list of at most R out-neighbors (usually 32 to 64), while in-neighbors are not stored. We will use the Vamana graph from DiskANN [9], which all three research papers build on. A search is a greedy beam search from a fixed entry point. It keeps the L closest nodes found so far, expands the closest unexpanded one by reading its neighbor list and computing the distance from the query to each neighbor, and stops when the whole list has been expanded. An insert of a point p searches for p with a larger beam, prunes the visited nodes with DiskANN's RobustPrune heuristic to pick p's out-neighbors, and then adds a reverse edge to p in each of those neighbors' lists, pruning any list that grows past R. A search only reads, while an insert reads one search path and writes at most $R + 1$ neighbor lists.

Deletion is where the three papers differ. Marking p as deleted is a single flag, but the edges pointing to p now lead nowhere and the paths through p are lost, so the graph around p has to be repaired. Since in-neighbors are not stored, even finding those edges can be expensive.

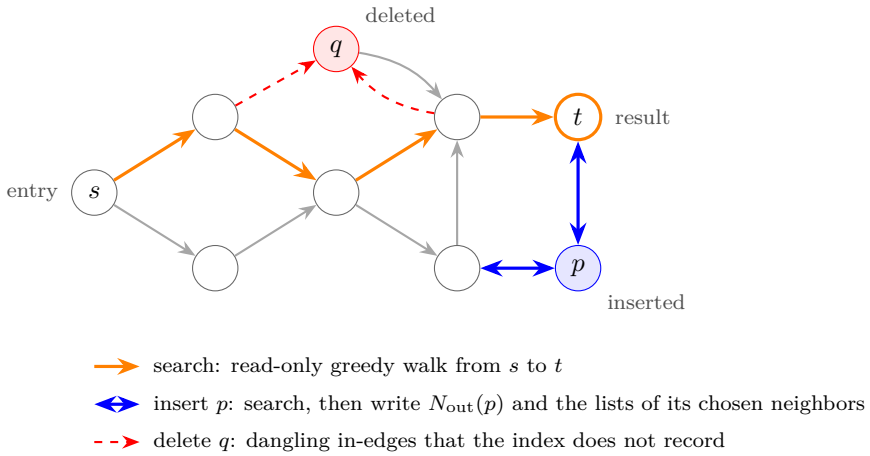


Figure 1: Which neighbor lists each operation touches. The search reads along one path, the insert writes the lists of p and its chosen neighbors, and the delete leaves edges into q from in-neighbors that the index does not record.

Strategy	Source	When the graph is repaired	What a delete of p does
Batched global consolidation	FreshDiskANN [8]	Periodically, by a scan over every node	Sets a tombstone flag on p
Repair at delete time	IP-DiskANN [10]	Immediately, inside the delete	Searches for x_p to approximate $N_{\text{in}}(p)$, rewrites about $\lvert N_{\text{in}}(p)\rvert + cR$ lists ($c = 3$), and unlinks p
Repair during later searches, with slot reuse	CleANN [11]	Lazily, by later searches and inserts that pass p	One compare-and-swap on p's status word

Table 1: The three deletion strategies we will implement. All three share FreshDiskANN's search and insert; they differ in who and when to repair the graph around a deleted point.

FreshDiskANN [8] is the base design: a delete only sets a tombstone, and a periodic consolidation pass scans the whole graph, reconnects the neighbors of every tombstoned node, and frees it. IP-DiskANN [10] and CleANN [11] are two independent 2025 follow-ups that keep FreshDiskANN's search and insert, and replace only the consolidation step. IP-DiskANN repairs eagerly inside the delete, by running a search to approximate p's in-neighbors and patching their lists. CleANN repairs lazily: the delete is one compare-and-swap, searches and inserts that later pass p run FreshDiskANN's per-node consolidation on the fly, and after a few such repairs p's slot is reused for a new point. Figure 1 shows which lists each operation touches and Table 1 summarizes the three designs.

The parallelism available is clear when the graph is static. Searches on an unchanging graph are independent, so a query batch is perfectly parallel across threads, and distance computations vectorize well since each one is a dot product over d floats repeated for every neighbor. Insertions of points that are far apart in vector space write disjoint neighbor lists, so they could run concurrently, and consolidation is also a data-parallel scan. What we want to find out is how much of these parallelism is still left when the three operations run at the same time on one shared graph.

The Challenge

Searching a static graph index in parallel is easy, since the queries are independent. The difficulty comes from running updates at the same time on a shared, pointer-based structure whose output quality can only be measured statistically. We expect to learn how lock granularity, the placement of repair work, and memory behavior all interact with each other, and which of them limits scaling on a real machine.

Every insert depends on the graph left by earlier inserts, and CleANN shows that the recall of FreshVamana varies substantially with insertion order [11], so the concurrent insertions' order changes the resulting index and its recall. Nearby points inserted at the same time search through the same region and add reverse edges to the same neighbor lists, which can be a lost-update race if the lists are not protected. Neighbor lists are read by every search and written by inserts and repairs, and the entry point and the hub nodes near it are on almost every path. With per-node reader-writer locks, those few nodes are contended, and their lock words bounce between cores even when nobody is writing. A global lock avoids that, but also serializes all updates against all searches. Optimistic reads avoid read-side writes entirely, but then have to detect concurrent modification and decide when a deleted node's slot can be reused while a reader may still be looking at it.

The traversal is a pointer-chasing process. Each hop reads one neighbor list and then up to R vectors scattered through memory, and the addresses of the next hop depend on these loads, so hardware prefetching does not help across hops. The vectors alone take 512 MB for SIFT1M, and about 4 GB for either GIST1M or a 10M slice of MS-Turing, far more than the last-level cache, so almost every vector load is a DRAM access, while a 128-dimensional distance is only 128 multiply-adds. The workload is memory-latency bound. The only locality is around the entry point, which is also where the write contention is. Work per operation also varies widely. A search may expand a handful of nodes or several hundred, and a delete costs anything from one compare-and-swap (CleANN) to a search plus a few hundred list writes (IP-DiskANN) to a scan of the whole index (FreshDiskANN), so threads in a mixed workload are poorly balanced.

Each repair strategy has a different cost under concurrency. FreshDiskANN's consolidation pass reads every list and rewrites a lot of them, so it either stops the world or runs as a long-lived writer. CleANN measured more than a 10× throughput drop in FreshVamana during consolidation [11]. IP-DiskANN's delete writes on the order of $|N_{\text{in}}(p)|+cR$ lists, and the write set is not bounded since in-degree is not bounded. The paper does not describe a concurrency protocol and was evaluated with 16 threads [10], so our concurrent version will be our own design. CleANN makes the delete trivial, but turns searches into writers whenever they pass a tombstone. Although the paper does not explicitly say so, we suspect that this is part of why CleANN's search throughput scales to about 20× at 224 threads while inserts reach about 63× [11]. Testing that is one of our analysis questions.

Resources

We are writing the index from scratch in C++, using the standard library only for threads and atomics; the reader-writer locks, the index, and the graph-repair logic are our own code. Data loading, brute-force ground truth, and plots will be Python. We will read the open-source DiskANN and ParlayANN [5] code and the authors' Rust implementation of CleANN as references and use

them as sanity checks on recall and single-thread speed, but we will not build on them.[1].

The three deletion strategies are reproductions of published algorithms: consolidation from FreshDiskANN (Algorithm 4 there), the in-place delete and cleanup from IP-DiskANN (Algorithms 5 and 6), and on-the-fly consolidation with slot reuse from CleANN (Algorithms 5 to 10). The synchronization comparison is our own. CleANN reuses FreshDiskANN's per-list reader-writer locks and IP-DiskANN does not describe its concurrency control, so the global-lock, per-node-lock, and optimistic-read variants, and the concurrent version of IP-DiskANN's delete, have no published design to follow. The workload generator and measurement harness are also ours,, since the three papers' drivers are tied to their own codebases and none of them measures recall and throughput under concurrent updates at the same time. The update runbooks (sliding window, expiration time, clustered) come from the NeurIPS'23 Big-ANN streaming track [6]. For reader-writer locks, optimistic synchronization, and memory reclamation, we will rely on Herlihy et al. [4].

During development, we will use SIFT1M (128-dimensional) and GIST1M (960-dimensional) from ANN-Benchmarks [3], both with exact ground truth. The main experiments use 1M to 10M prefixes of MS-Turing (100-dimensional) from Big-ANN-Benchmarks, since both 2025 papers use it. Ground truth after each update round is computed by brute force on the live set, on a sample of the queries if the full set is too slow.

We will develop on the GHC machines. Hardware counters will come from Linux `perf`.

What we still need: a machine with more cores than the GHC machines for the scaling study. We would like to ask the course staff whether one is available, and if not, we will use a many-core cloud instance as our fallback.

Goals and Deliverables

Our goal is to build a concurrent, in-memory graph-based vector index in C++ and study how synchronization granularity and deletion-repair strategies affect search quality and multicore scalability. The main deliverables are a working implementation, a reproducible benchmark suite, and an experimental analysis.

Plan to Achieve

1. Implement a sequential graph-based index supporting approximate top-k search, insertion, and deletion, validated against brute-force ground truth.

2. Implement two synchronization schemes: a global reader-writer lock and per-node reader-writer locks, supporting concurrent searches and updates.

3. Implement three deletion-repair strategies: batched consolidation (FreshDiskANN), immediate repair (IP-DiskANN), and search-time repair with slot reuse (CleANN).

4. Evaluate on SIFT1M, GIST1M, and 1M–10M subsets of MS-Turing using static search, sliding-window updates, concurrent mixed workloads, and expiration-based and clustered updates.

We will measure recall, search and update throughput, latency, and scaling from one thread to the available physical cores. Our goal is to achieve measurable multicore speedup without

[1] https://github.com/microsoft/DiskANN, https://github.com/cmuparlay/ParlayANN, https://github.com/SylviaZiyuZhang/cleann-rust

substantial recall degradation, and determine which synchronization and repair strategies perform best under different update ratios and thread counts.

Our analysis will answer three questions: where deletion repair should be placed (a global batch, the delete itself, or later searches) for throughput and recall under concurrency; which lock granularity wins at each update ratio; and whether lock contention or memory latency limits scaling. We will also test whether CleANN's weaker search scaling comes from searches acting as writers.

Hope to Achieve

If time permits, we hope to implement optimistic lock-free reads, explore NUMA-aware memory placement if we obtain a multi-socket machine, or evaluate the index on real RAG embeddings such as OpenScholar [1].

Fallback Goals

If implementation takes longer than expected, we will prioritize concurrent insertions and searches with both synchronization schemes and two deletion-repair strategies. We will reduce dataset sizes and workload coverage while retaining correctness tests, recall measurements, and multicore performance comparisons.

Platform Choice

We will implement the index in C++ and evaluate it on a shared-memory multicore CPU system.

C++ is suitable for our task as it provides low-level control over memory layout, synchronization, and thread execution, which is essential for implementing our own locks and graph-repair mechanisms. Besides, both of us are familiar with C++ which reduces the time to familiarize ourselves with the programming language and helps us stay focused on the main tasks.

A multicore CPU is well suited to this workload because graph-based vector search involves irregular memory accesses and pointer chasing, while concurrent insertions and deletions modify shared graph structures. A shared-memory system allows us to directly study contention between searches and updates, compare synchronization granularity, and measure scalability as thread counts increase. A GPU is a poor fit for the same reasons: each search step is a data-dependent pointer chase, and updates make fine-grained changes to shared state, which is also why all three papers we build on evaluate on multicore CPUs [8, 10, 11]. We will develop on the GHC machines, whose 8-core CPUs are enough for correctness work and the first scaling curves, and run the scaling study on a machine with more cores (see Resources).

Schedule

- Week 1 (Nov 9–15): Set up the C++ environment, load the benchmark datasets, and write the brute-force ground-truth tool and the recall test. Implement the sequential graph index with search, insertion, and deletion, and establish baseline performance.
- Week 2 (Nov 16–22): Implement the global and per-node reader-writer locks and the workload generator (static, sliding-window, mixed, expiration-time, and clustered runbooks). Implement the first deletion-repair strategy (batched consolidation). Test concurrent searches and

insertions, measure the first scaling curve, and write the milestone report, due Monday, Nov 23.

- Week 3 (Nov 23–29): Implement the remaining two deletion-repair strategies (immediate repair and search-time repair with slot reuse) under per-node locks. Thanksgiving falls in this week, so we schedule implementation only and no experiments.
- Week 4 (Nov 30–Dec 6): Integrate all three strategies with both synchronization schemes and run the scalability experiments across thread counts and update ratios on SIFT1M, GIST1M, and feasible MS-Turing subsets, with hardware counters. Analyze recall, throughput, latency, and performance bottlenecks.
- Week 5 (Dec 7–11): Write the final report (due Thursday, Dec 10) and prepare the poster (session Friday, Dec 11).

References

[1] Akari Asai, Jacqueline He, Rulin Shao, Weijia Shi, Amanpreet Singh, Joseph Chee Chang, Kyle Lo, Luca Soldaini, Sergey Feldman, Mike D'Arcy, David Wadden, Matt Latzke, Minyang Tian, Pan Ji, Shengyan Liu, Hao Tong, Bohao Wu, Yanyu Xiong, Luke Zettlemoyer, Graham Neubig, Dan Weld, Doug Downey, Wen-tau Yih, Pang Wei Koh, and Hannaneh Hajishirzi. OpenScholar: Synthesizing scientific literature with retrieval-augmented LMs. *arXiv preprint arXiv:2411.14199*, 2024.

[2] Akari Asai, Zexuan Zhong, Danqi Chen, Pang Wei Koh, Luke Zettlemoyer, Hannaneh Hajishirzi, and Wen-tau Yih. Reliable, adaptable, and attributable language models with retrieval. *arXiv preprint arXiv:2403.03187*, 2024.

[3] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems*, 87, 2020.

[4] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2nd edition, 2020.

[5] Magdalen Dobson Manohar, Zheqi Shen, Guy E. 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 Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2024.

[6] Harsha Vardhan Simhadri, Martin Aumüller, Amir Ingber, Matthijs Douze, George Williams, Magdalen Dobson Manohar, Dmitry Baranchuk, Edo Liberty, Frank Liu, Ben Landrum, Mazin Karjikar, Laxman Dhulipala, Meng Chen, Yue Chen, Rui Ma, Kai Zhang, Yuzheng Cai, Jiayang Shi, Yizhuo Chen, Weiguo Zheng, Zihao Wan, Jie Yin, and Ben Huang. Results of the Big ANN: NeurIPS'23 competition. *arXiv preprint arXiv:2409.17424*, 2024.

[7] Harsha Vardhan Simhadri, George Williams, Martin Aumüller, Matthijs Douze, Artem Babenko, Dmitry Baranchuk, Qi Chen, Lucas Hosseini, Ravishankar Krishnaswamy, Gopal Srinivasa, Suhas Jayaram Subramanya, and Jingdong Wang. Results of the NeurIPS'21 challenge on billion-scale approximate nearest neighbor search. In *Proceedings of the NeurIPS 2021 Competitions and Demonstrations Track, PMLR 176*, 2022.

[8] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. FreshDiskANN: A fast and accurate graph-based ANN index for streaming similarity search. *arXiv preprint arXiv:2105.09613*, 2021.

[9] Suhas Jayaram Subramanya, Devvrit, Rohan Kadekodi, Ravishankar Krishaswamy, and Harsha Vardhan Simhadri. DiskANN: Fast accurate billion-point nearest neighbor search on a single node. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[10] Haike Xu, Magdalen Dobson Manohar, Philip A. Bernstein, Badrish Chandramouli, Richard Wen, and Harsha Vardhan Simhadri. In-place updates of a graph index for streaming approximate nearest neighbor search. *arXiv preprint arXiv:2502.13826*, 2025.

[11] Ziyu Zhang, Yuanhao Wei, Joshua Engels, and Julian Shun. CleANN: Efficient and robust full dynamism in graph-based approximate nearest neighbor search. In *Proceedings of the 38th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2026. arXiv:2507.19802.