Increasing Throughput with Concurrency Control for ANN Indexes

Collin Drake
University of Colorado Boulder
collin.drake@colorado.edu

Abstract

Implementing thread-safety in ANN indexes presents a variety of design choices. Ultimately, the tradeoffs converge to two axes: locking granularity and conflict resolution. In IVFFlat, for example, locking can be applied to each bucket; while effective, this approach is coarse-grained. Conversely, HNSW allows for locking at the finer granularity of a node's neighbor list. In both architectures, conflicts may be handled optimistically or pessimistically. In this paper, we implement and evaluate the cross-product of these strategies.

1. Introduction

As vector databases increasingly support streaming workloads, the need for concurrent queries against continuously updated indexes has become apparent [1]. While approximate nearest neighbor (ANN) search has been extensively studied under the assumption of a static index [2], the concurrency control mechanisms required for dynamic workloads remain underexplored [3]. Existing systems either sacrifice read-write concurrency entirely or rely on batch processing strategies incompatible with streaming insertions. This paper systematically examines the design space of concurrency control for ANN indexes, focusing on the interplay between locking granularity and conflict resolution strategies.

Existing approaches to parallel ANN index construction fall short of these requirements. FAISS, a well-known example of fine-grained locking, utilizes this locking strictly for parallel index construction (protecting concurrent writers), rather than for isolation between readers and writers [4]. Consequently, it does not support simultaneous search and update. Furthermore, while FAISS exploits thread-level parallelism to batch-process search queries, the search path itself is completely lock-free [5].

ParlayANN takes a different approach, offering two strategies for parallel construction. Its deterministic batch construction uses prefix sums and parallel primitives to process points in a fixed order, ensuring reproducible results across runs. For incremental updates, it employs lock-free batch insertion using compare-and-swap operations, allowing threads to modify neighbor lists without acquiring locks [6]. However, neither technique is well-suited for online systems. Prefix sum-based construction requires the entire batch of points to be known upfront; offsets must be computed before any insertions can proceed, making it fundamentally incompatible with a stream of arriving vectors. Similarly, batch conflict scheduling assumes all operations are available for analysis before execution, enabling the scheduler to order them to minimize conflicts. In an online setting, there is no batch to analyze.

JVector implements lock-free concurrent construction but is not benchmarked on standard suites, making performance comparisons difficult [7]. And neither FAISS' parallel construction nor ParlayANN's batch strategies address the online read-write concurrency problem we examine.

This gap motivates our systematic exploration of the concurrency control design space for online ANN indexes. We identify two key dimensions along which strategies may vary: locking granularity (coarse-grained at the partition or bucket level vs. fine-grained at the individual neighbor list level) and conflict resolution policy (optimistic with validation and retry vs. pessimistic with blocking) [8]. The cross-product of these dimensions yields four distinct strategies. We implement each strategy for both IVFFlat and HNSW indexes and evaluate their performance under mixed read-write workloads. Our evaluation provides the first empirical comparison of concurrency control approaches for streaming vector search.

2. Methodology

We evaluate the performance of our concurrency control strategies using a standard ANN benchmark dataset. Specifically, we use the Fashion-MNIST dataset ($d = 784$), measuring performance under a mixed workload of search and insertion operations.

2.1. Dataset and Workload

The primary workload consists of a concurrent mix of document insertions and approximate nearest neighbor searches. The write ratio ramps linearly from 1% to 5% as thread count increases from 2 to 16, so that contention pressure grows alongside parallelism. We vary the number of concurrent worker threads from 2 to 16 to observe the scaling behavior of each index implementation.

Standard database benchmarks model production workloads as overwhelmingly read-heavy. YCSB Workload B (95% read, 5% update) and Workload D (95% read, 5% insert) represent the read-mostly tier of the Yahoo! Cloud Serving Benchmark suite [9], and even the "update-heavy" Workload A allocates only 50% of operations to writes. Production vector databases skew further toward reads: ingestion is typically batched or periodic, while queries arrive continuously. Our 1–5% write range falls within this production regime. However, our goal is to stress concurrency control mechanisms, not to replicate a particular deployment profile. By ramping the write ratio across thread counts, we ensure that write-write and read-write conflicts increase with parallelism, exposing scaling bottlenecks that a fixed low write ratio would mask at higher thread counts.

We use the Fashion-MNIST dataset, utilizing the Euclidean distance metric. The dataset contains 60,000 training vectors and 10,000 query vectors, with a dimensionality of 784.

2.2. Experimental Environment

All experiments were conducted on a Lenovo Legion Pro 7i 16IAX10H with an Intel Core Ultra 9 275HX CPU (24 total cores: 8P+16E) and 32 GiB RAM. CPU-only measurements are reported for throughput and recall; GPU acceleration was excluded from the primary comparisons to keep index implementations on a consistent execution target. The benchmarking harness is implemented in Python and invokes the core C++ index implementations through bindings to minimize orchestration overhead.

3. Results

3.1. Throughput

Incidentaly, the lower a dataset's throughput, the higher is cluster density.

Qdrant was omitted from the plotted throughput comparison because its performance was substantially lower and did not scale with thread count in this workload (2 threads: 561 ops/s, 4: 537 ops/s, 8: 571 ops/s, 16: 530 ops/s).

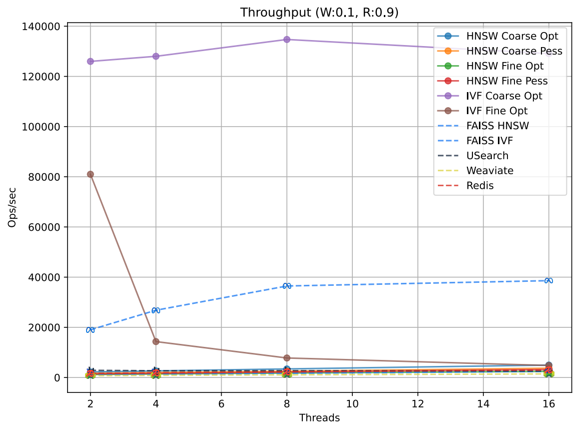

Figure 2: SIFT-128

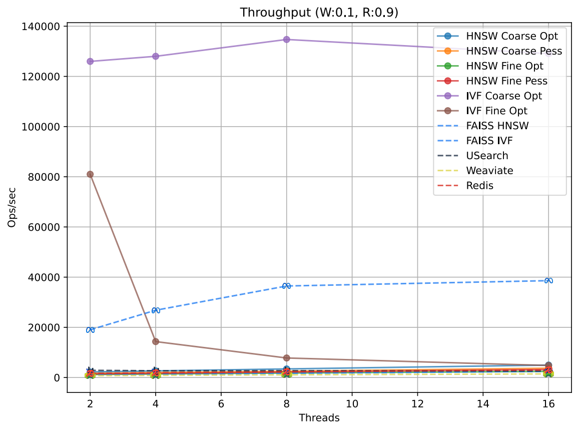

Figure 3: Fashion-MNIST-784

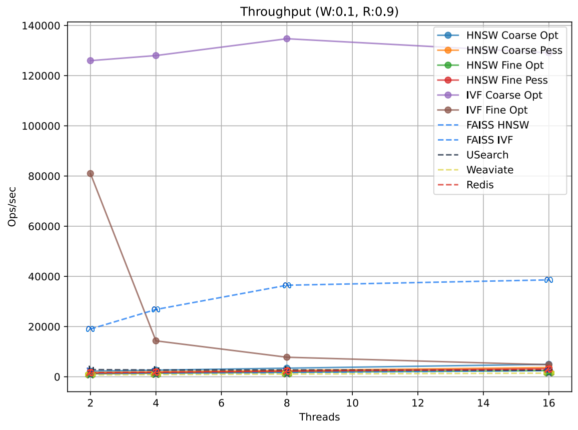

Figure 4: GloVe-100

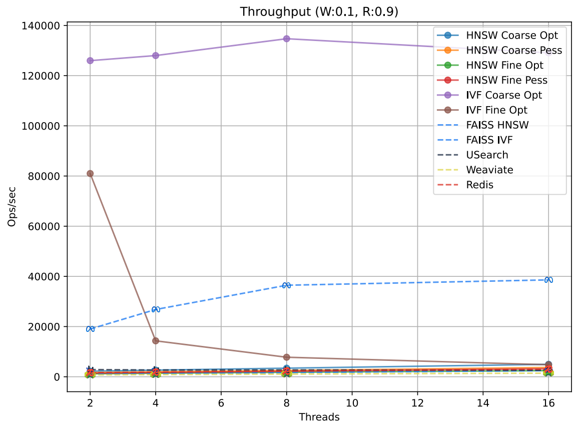

Figure 5: GloVe-25

Figure 1: Throughput scaling with increasing numbers of threads for each index type, across all benchmark datasets.

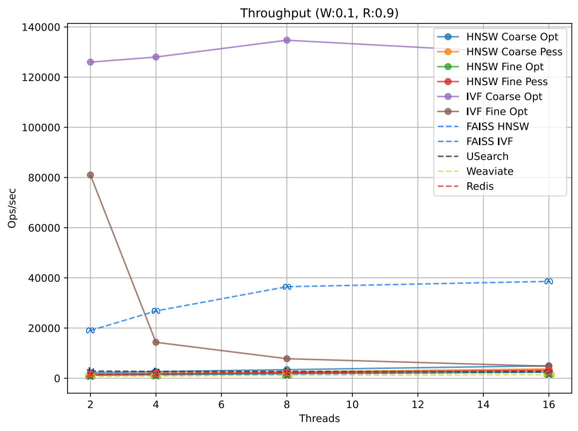

Figure 7: GIST-960

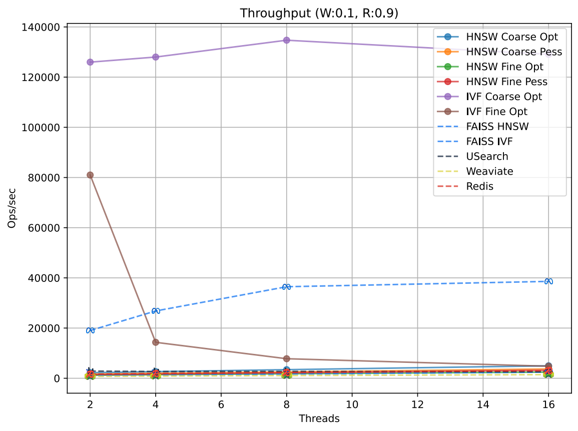

Figure 8: NYTimes-256

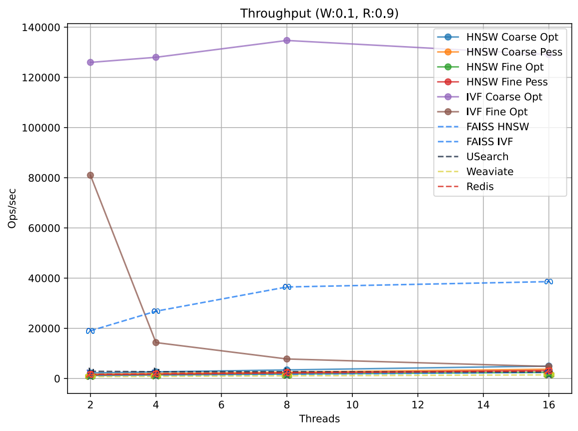

Figure 9: MNIST-784

Figure 6: Throughput scaling with increasing numbers of threads for each index type, across all benchmark datasets.

4. Discussion

- ***Why hasn't this been done before?*** The predominant paradigm has been offline construction followed by read-only serving, so reader-writer concurrency at the index level simply never arose. Systems that do accept writes typically side-step the problem architecturally: mutable segments are flushed to read-only storage on a rolling basis, and searches hit only sealed data [10]. Streaming workloads driven by retrieval-augmented generation [11] have only recently made in-place concurrent update a practical requirement.

- ***Why does HNSW degrade more under concurrent writes than IVF?*** The answer lies in a structural asymmetry between the two index types. HNSW is a navigable small-world graph: its search correctness depends on the graph remaining well-connected across layers [12]. During insertion, a new node is wired into the graph by selecting neighbors via a heuristic and then back-linking those neighbors to the new node [12]. Under concurrent writes, these two steps are not atomic. A racing writer can observe a partially-linked node, traverse a stale edge, or have its own neighbor list pruned before its back-links are established, any of which can leave the graph with weakly connected or entirely isolated nodes. Heuristics like deferred or batched pruning reduce the frequency of such breaks but cannot eliminate them: they trade recall loss for throughput, rather than recovering the full structural guarantee. IVF does not share this vulnerability. At the coarse quantizer level, cluster assignment is a read-only operation; inserting a vector into a cluster appends to a list and requires only a per-cluster lock [4]. Concurrent writers targeting different clusters are entirely independent, and even writers within the same cluster contend only on a flat append structure with no graph invariant to preserve. The degradation seen in HNSW throughput and recall under high write concurrency is therefore not merely an implementation artifact; it reflects a fundamental tension between the graph connectivity invariant that makes HNSW fast and the atomicity that concurrent mutation requires.

5. Conclusion

Bibliography

[1] S. Gong, H. Sun, Z. Fang, and L. Liu, "VStream: A Distributed Streaming Vector Search System," *Proceedings of the VLDB Endowment*, vol. 18, no. 6, 2025, doi: 10.14778/3725688.3725692.

[2] L. Ma *et al.*, "A Comprehensive Survey on Vector Database: Storage and Retrieval Technique, Challenge." [Online]. Available: https://arxiv.org/abs/2310.11703

[3] A. Singh, S. J. Subramanya, R. Krishnaswamy, and H. V. Simhadri, "FreshDiskANN: A Fast and Accurate Graph-Based ANN Index for Streaming Similarity Search." [Online]. Available: https://arxiv.org/abs/2105.09613

[4] M. Douze *et al.*, "The Faiss Library." [Online]. Available: https://arxiv.org/abs/2401.08281

[5] Meta Platforms, Inc., "FAISS: IndexHNSW Implementation." [Online]. Available: https://github.com/facebookresearch/faiss/blob/main/faiss/IndexHNSW.cpp

[6] M. D. Manohar *et al.*, "ParlayANN: Scalable and Deterministic Parallel Graph-Based Approximate Nearest Neighbor Search Algorithms." [Online]. Available: https://arxiv.org/abs/2305.04359

[7] DataStax, "JVector: The Most Advanced Embedded Vector Search Engine." [Online]. Available: https://github.com/datastax/jvector

[8] H. T. Kung and J. T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems*, vol. 6, no. 2, pp. 213–226, 1981, doi: 10.1145/319566.319567.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, in SoCC '10. ACM, 2010, pp. 143–154. doi: 10.1145/1807128.1807152.

[10] J. Wang *et al.*, "Milvus: A Purpose-Built Vector Data Management System," in *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*, 2021, pp. 2614–2627. doi: 10.1145/3448016.3457550.

[11] P. Lewis *et al.*, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in *Advances in Neural Information Processing Systems*, 2020, pp. 9459–9474. [Online]. Available: https://arxiv.org/abs/2005.11401

[12] Y. A. Malkov and D. A. Yashunin, "Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 4, pp. 824–836, 2020, doi: 10.1109/TPAMI.2018.2889473.