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. While approximate nearest neighbor (ANN) search has been extensively studied under the assumption of a static index [1], the concurrency control mechanisms required for dynamic workloads remain underexplored. 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. 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 [2].

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 [3]. 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. The system must handle each operation as it arrives while maintaining consistency with concurrent operations already in flight.

JVector implements lock-free concurrent construction but is not benchmarked on standard suites, making performance comparisons difficult [4]. And neither FAISS's 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). 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 (10%) and approximate nearest neighbor searches (90%). This ratio models a read-heavy application with a continuous stream of updates, typical of many real-time vector search scenarios. We vary the number of concurrent worker threads from 1 to 16 to observe the scaling behavior of each index implementation.

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. Baselines

To contextualize the performance of our fine-grained locking implementations, we compare against several state-of-the-art open-source vector search libraries:

- **FAISS**: A library for efficient similarity search and clustering of dense vectors. We evaluate both its HNSW and IVF implementations. Note that while FAISS supports parallel construction, its standard search index is not designed for concurrent read/write operations without external locking.
- **USearch**: A smaller, header-only vector search library generally known for good performance and simplicity.
- **Weaviate**: An open-source vector database that manages concurrent access, offering a system-level comparison point.

2.3. Experimental Environment

All experiments were conducted on a Lenovo Legion Pro 7 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.

Google Colab was not used for primary CPU benchmark reporting because runtime hardware and scheduling are not fixed across sessions, which reduces reproducibility for multi-thread scaling comparisons.

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).

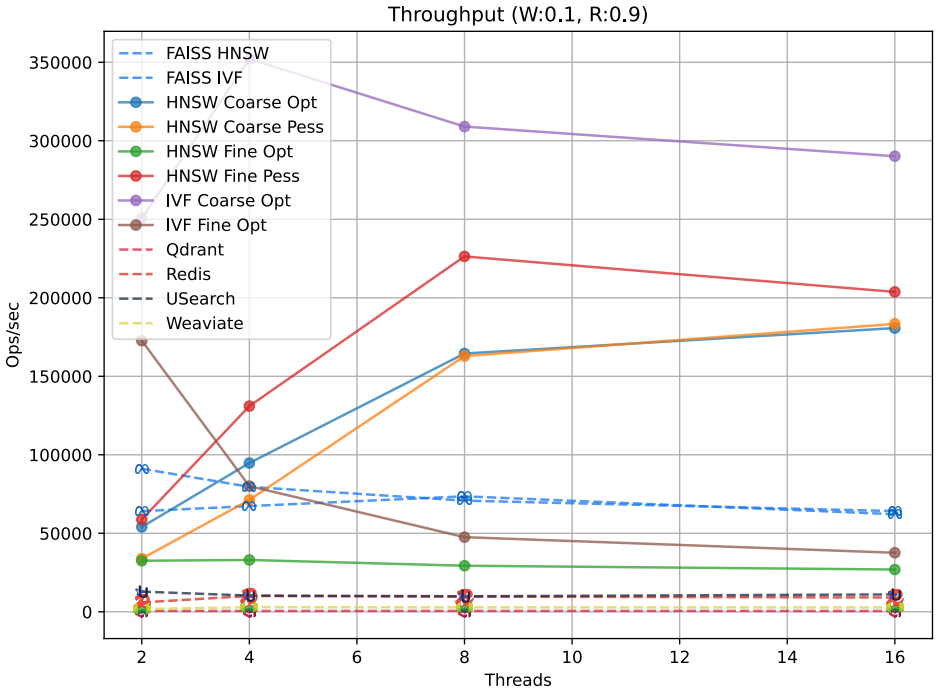

Figure 1: Throughput scaling with increasing numbers of threads for each index type.

4. Discussion

- ***Why hasn't this been done before?*** That's an excellent question. Techniques as trivial for database concurrency control as these should have been implemented long ago. Well, one possiblity is that for most applications these techniques are unnecessary; most ML workflows simply don't require this level of concurrency control. Another possiblity is that these indexes have had *relatively* short times spent in the research limelight.

5. Conclusion

Bibliography

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

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

[3] 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

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