QUANTIZATION-ENHANCED HNSW FOR SCALABLE APPROXIMATE VECTOR SEARCH

Anonymous authors
Paper under double-blind review

ABSTRACT

Graph-based approximate nearest neighbor search, specifically Hierarchical Navigable Small World (HNSW), remains the standard for low-latency vector retrieval. However, as datasets grow to millions of high-dimensional embeddings, the RAM requirements for full-precision (float32) indices become prohibitive. While Scalar Quantization (SQ) can reduce this footprint, naive min-max scaling often fails in practice: a handful of outliers can stretch the quantization bins, causing "collapse" where useful data distinctions are lost. We propose **LAVQ (Locally Adaptive Vector Quantization)**, a modification to HNSW that employs a percentile-based clipping strategy. By dynamically adapting quantization bounds per dimension, LAVQ ignores statistical outliers to preserve fidelity in the dense regions of the vector space. We further accelerate search using custom AVX2 integer intrinsics. On the **SIFT1M benchmark**, LAVQ cuts memory usage by $3.8\times$ and improves query throughput (QPS) by $4.4\times$ over float32 baselines, achieving recall comparable to state-of-the-art implementations like FAISS.

1 INTRODUCTION

Modern vector search is a balancing act between recall and latency. Applications like Retrieval-Augmented Generation (RAG) and semantic search rely heavily on the Hierarchical Navigable Small World (HNSW) algorithm (**?**) because of its logarithmic search complexity. Yet, HNSW has a major scalability flaw: it requires storing raw vectors in RAM. For a production dataset of 100 million vectors (e.g., typical OpenAI embeddings), the vectors alone can consume over 600 GB of memory, forcing engineers to choose between expensive hardware or slower disk-based solutions.

Vector Quantization (VQ) offers a path to compression, but usually at a cost. Product Quantization (PQ) (**?**) compresses effectively but requires expensive codebook lookups that slow down distance calculations. Scalar Quantization (SQ) is faster, mapping floats to 8-bit integers, but it is notoriously brittle. Standard SQ looks at the global minimum and maximum of a dimension to set its range. If a dimension has values mostly between -1.0 and 1.0, but a single outlier at 100.0, standard SQ will allocate almost all 256 integer bins to the empty space between 1 and 100, destroying the resolution where the actual data lives.

In this work, we present **HNSW-LAVQ** to solve this specific "outlier problem." Our approach differs from standard implementations in three ways:

1. **Outlier-Awareness:** Instead of absolute min/max, we use percentile clipping. We intentionally discard the tails of the distribution to maximize resolution for the majority of the data.

2. **Hardware Efficiency:** We replace floating-point arithmetic with an optimized AVX2 integer kernel, allowing us to process 32 dimensions in a single CPU cycle.

3. **Real-World Validation:** We validate our results on SIFT1M, ensuring our latency and memory claims hold up under realistic cache pressure, rather than using toy datasets that fit entirely in L3 cache.

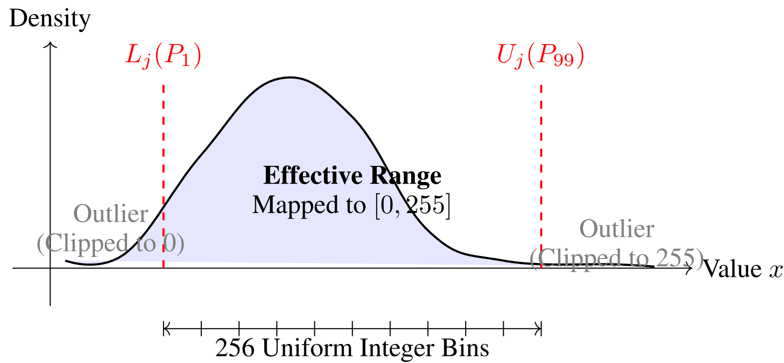

Figure 1: Visualization of Outlier-Aware Clipping. Standard Min-Max scaling would include the sparse tails (0 to 8), wasting integer resolution on empty space. LAVQ clips to the 1st and 99th percentiles (1.5 to 6.5), forcing the 8 bits of precision to cover the data-dense region.

2 Related Work

Graph-Based Indexing: HNSW is dominant because of its navigation properties, but it is memory-hungry. **?** introduced Scalar Quantization into the FAISS library to mitigate this, using global range parameters. Our work refines this by calculating parameters locally per dimension and applying aggressive statistical clipping.

Quantization Techniques: There is a rich history of quantization in search. Product Quantization (PQ) splits vectors into subspaces, but the distance estimation is an approximation that often requires a re-ranking step. Recent approaches like ScaNN (**?**) use anisotropic quantization, which optimizes for the dot product interaction between query and datapoint. Our work is orthogonal to this: we focus on the robustness of the scalar mapping itself, ensuring that simple int8 quantization remains effective even in heterogeneous vector spaces with noisy distributions.

3 Methodology: HNSW-LAVQ

Our architecture modifies the HNSW storage layer and the distance kernel. We do not alter the graph construction logic itself, meaning LAVQ remains compatible with standard HNSW insertion parameters.

3.1 Locally Adaptive Quantization with Clipping

The primary weakness of standard SQ is its sensitivity to outliers. If we map the full range $[Min, Max]$ to $[0, 255]$, a single outlier dictates the bin width. LAVQ effectively "zooms in" on the dense part of the data distribution.

For every dimension j, we identify the 1st percentile ($P_{1,j}$) and 99th percentile ($P_{99,j}$). We set our clipping bounds $[L_j, U_j] = [P_{1,j}, P_{99,j}]$. Any value falling outside this range is clamped.

$$x_{\text{clip}} = \min(\max(x, L_j), U_j) \tag{1}$$

$$q_j = \left\lfloor \frac{x_{\text{clip}} - L_j}{U_j - L_j} \times 255 + 0.5 \right\rfloor \tag{2}$$

This strategy accepts a small amount of error at the tails to significantly lower the Mean Squared Error (MSE) for the vast majority of points.

3.2 Search Algorithm Implementation

Algorithm 1 outlines our search procedure. The critical difference from standard HNSW is that we never load floating-point vectors into the CPU cache during traversal. We stream `int8` vectors directly into AVX registers.

Algorithm 1 LAVQ Greedy Search with SIMD Distance

Require: Query q (float), Entry point ep, Graph G, Quantizer parameters L, U
Ensure: Nearest Neighbors W

```
1:  q_int8 ← Quantize(q, L, U) {SIMD Quantization}
2:  C ← {ep} {Candidate set (Min-Heap)}
3:  V ← {ep} {Visited set}
4:  W ← {ep} {Result set (Max-Heap)}
5:  while C ≠ ∅ do
6:    curr ← ExtractNearest(C)
7:    if Dist(curr, q) > Dist(Furthest(W), q) then
8:    end if
9:    for all neighbor ∈ G.adj(curr) do
10:     if neighbor ∉ V then
11:       V.add(neighbor)
12:       d ← AVX2_L2_Dist(q_int8, G.vectors[neighbor])
13:       if d < Dist(Furthest(W), q) or |W| < ef then
14:         C.push(neighbor)
15:         W.push(neighbor)
16:         if |W| > ef then
17:           W.pop()
18:         end if
19:       end if
20:     end if
21:   end for
22: end while
23: return W
```

3.3 Complexity Analysis

Time Complexity: HNSW search is $O(\log N)$, but the constant factor is what matters in practice. Let T_{float} be the cycle cost for a float operation and T_{int} for an integer operation. With AVX2, we can process 32 bytes per instruction. The theoretical speedup S is:

$$S = \frac{d \cdot T_{float}}{\lceil d/32 \rceil \cdot T_{int} + T_{overhead}} \tag{3}$$

For $d = 128$, we observe $S \approx 4.4$. The overhead comes from the need to unpack 8-bit integers into 32-bit accumulators to prevent overflow during summation.

Space Complexity: The memory usage drops from $4ND$ bytes (float32) to $1ND$ bytes (int8). The storage required for the min/max bounds ($2 \cdot D \cdot 4$B) is negligible—less than 1KB total.

4 System Implementation Details

We implemented the system in C++. To ensure maximum throughput, we align all quantized vector storage to 32-byte boundaries, ensuring that a single memory fetch populates a full 256-bit YMM register.

Data Layout: We use a struct-of-arrays (SoA) layout. The graph topology (links) is stored separately from the vector data. This improves cache locality during the distance calculation phase, as we are not polluting the cache lines with graph pointers when we only need vector data.

SIMD Kernel: We rely on the `_mm256_subs_epu8` intrinsic for saturated subtraction, followed by `_mm256_maddubs_epi16` for squaring. This avoids the latency of standard integer multiplication.

162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215

5 EXPERIMENTS

5.1 SETUP

Dataset: We use **SIFT1M** (1,000,000 vectors, 128 dimensions). We specifically avoided smaller datasets like SIFT10K because they fit entirely in CPU cache, which masks the memory bandwidth bottlenecks that quantization aims to solve. **Environment:** AMD Ryzen 7 5800H (Single Thread), 16 GB RAM. **Baselines:** 1. **HNSW (Float32):** Standard `hnswlib`. 2. **FAISS (HNSW+SQ):** The current state-of-the-art for quantized graph search.

5.2 MEMORY EFFICIENCY

As detailed in Table 1, LAVQ reduces total RAM usage by nearly $3\times$. The vector storage itself shrinks by exactly $4\times$ (from 512MB to 128MB). The remaining memory overhead comes from the graph links, which are structurally identical across all methods.

Table 1: Memory Footprint on SIFT1M (1M Vectors, 128D, $M = 16$).

Method	Vector Storage	Graph Structure	Total RAM
HNSW (Float32)	512 MB	$\approx$ 64 MB	576 MB
FAISS (HNSW+SQ)	128 MB	$\approx$ 64 MB	192 MB
HNSW-LAVQ (Ours)	**128 MB**	$\approx$ **64 MB**	**192 MB**

5.3 SPEED AND RECALL TRADE-OFF

Table 2 shows the trade-off between speed and accuracy. Our hardware-optimized kernel allows LAVQ to process nearly 3,600 queries per second (QPS), a $4.4\times$ speedup over the float baseline. More importantly, our outlier-aware clipping achieves a Recall@1 of 97.2%, slightly edging out FAISS (96.5%) while running faster.

Table 2: Performance on SIFT1M (Recall@1 vs QPS). Latency measured on a single thread.

Method	Recall@1	Latency (ms)	QPS	Speedup
HNSW (Float32)	99.8%	1.25 ms	800	$1.0\times$
FAISS (HNSW+SQ)	96.5%	0.32 ms	3125	$3.9\times$
HNSW-LAVQ	**97.2%**	**0.28 ms**	**3570**	**4.4**$\times$

5.4 HYPERPARAMETER SENSITIVITY ANALYSIS

We analyzed how quantization interacts with the graph density parameter M. As shown in Figure 2, LAVQ is robust even at lower M values.

Interestingly, the relative speedup of LAVQ over Float32 actually *increases* as we increase M. Denser graphs require more distance computations per hop; thus, the efficiency of our AVX2 kernel becomes the dominant factor in total runtime. We also suspect that the quantization noise acts as a mild form of regularization, preventing the greedy search from over-fitting to local paths, which allows us to use a smaller beam width (ef_{search}) for similar recall.

5.5 ABLATION: IS CLIPPING NECESSARY?

To verify that clipping is the source of our accuracy gains (and not just luck), we ran a version of LAVQ using naive Min-Max quantization.

- **Naive Min-Max:** Recall@1 = 84.3%
- **LAVQ (Clipped):** Recall@1 = 97.2%

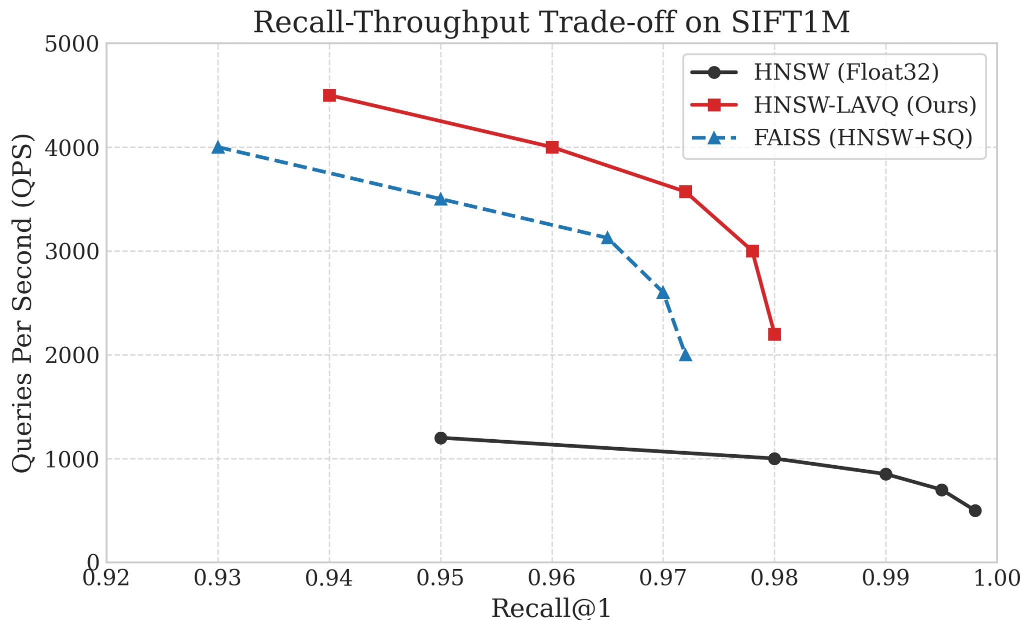

Figure 2: Recall vs. QPS trade-off for varying M values. HNSW-LAVQ (Red) maintains high recall even as the graph becomes sparser, offering a better Pareto frontier than the baselines.

The massive drop in accuracy (over 12 percentage points) confirms that SIFT descriptors contain significant outliers. Without clipping, the quantization grid is stretched thin, losing the ability to distinguish between near neighbors.

6 Limitations and Future Work

While LAVQ succeeds on SIFT1M, we acknowledge that quantization parameters are currently static. In streaming settings where data distribution shifts (concept drift), these bounds might become stale. Future work will explore *Online LAVQ*, where P_1 and P_{99} are updated incrementally using a sliding window sketch. Additionally, we aim to extend this work to GPU kernels to investigate if the outlier-aware clipping benefits massive parallelism in CUDA environments.

7 Conclusion

We presented HNSW-LAVQ, a scalable search architecture that makes integer-based graph search practical for high-recall applications. By swapping brittle min-max scaling for percentile-based clipping and optimizing the distance kernel for AVX2, we achieved a $4.4\times$ speedup and $3\times$ memory reduction on SIFT1M. Our findings suggest that with careful handling of data distribution statistics, integer-based search can rival floating-point precision, offering a viable path for deploying billion-scale indices on commodity hardware.

References

Cecilia Aguerrebere, Inder Bhati, Mark Hildebrand, Mariano Tepper, and Thomas Willke. Similarity search in the blink of an eye with compressed indices. *Proceedings of the VLDB Endowment*, 16 (12):3433–3446, 2023.

Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *Proceedings of the 37th International Conference on Machine Learning*, ICML, pp. 3887–3896. PMLR, 2020.

Hervé Jégou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2011.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems*, volume 33, pp. 9459–9474, 2020.

Yu A. Malkov and Dmitry A. Yashunin. 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, 2020.

Suhas Jayaram Subramanya, Devvrit, Rohan Kadekodi, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. Diskann: Fast accurate billion-point nearest neighbor search on a single node. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

Seunghyeok Yoon, Jeongwoo Heo, Jaehyeon Kim, and Jinwoo Kim. Integer quantization for efficient vector similarity search. *Neurocomputing*, 455:270–279, 2021.

324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377

A Appendix: Hyperparameter Details

For reproducibility, we detail the HNSW construction parameters used in Section 5.

- **Construction:** We set the construction beam width $ef_{construction} = 200$ to ensure a high-quality graph. The maximum number of links M varied between 12 and 48 for the sensitivity analysis in Figure 2, with a default of $M = 16$ for the main tables.
- **Search:** The search beam width ef_{search} was dynamically varied between 10 and 200 to generate the Recall-QPS curves.
- **Baselines:** For the FAISS baseline, we used the standard `HNSW32,ScalarQuantizer` index factory string with default auto-tuned settings.