Approaching Closed-Form Models for HNSW Hyperparameters at Scale [E]

Arijus Trakymas, Rachanon Petchoo, Kendrit Tahiraj, Sharva Darpan Thakur
{atraky2,petchoo2,ktahir2,sthaku25}@illinois.edu
University of Illinois Urbana-Champaign, Urbana, Illinois, USA

Abstract

Approximate Nearest Neighbor (ANN) algorithms, like Hierarchical Navigable Small World (HNSW), are significant in modern vector search and retrieval systems. While HNSW is one of the most popular choices due to its balance between recall, latency, and memory efficiency, challenges arise when trying to manually tune it's hyperparameters in real-world scenarios. Existing research typically evaluates these hyperparameters individually, or "one knob at a time". Our study conducts comprehensive, compound evaluations across both index construction and query-time hyperparameters and datasets. Using structured experiments over five benchmark datasets, we analyze trade-offs involving recall, latency, throughput, build time, and memory usage. Our findings model how multi-parameter tuning affects system performance and offers guidance for an optimal and pain-free HNSW deployment. We provide our evaluation code along with all our visualizations in a github repository here.

1 Introduction

Approximate Nearest Neighbor (ANN) algorithms are crucial for efficient vector search and retrieval tasks, powering modern applications such as recommendation systems, computer vision, and natural language processing [8]. Among various ANN techniques, Hierarchical Navigable Small World (HNSW) [5] stands out due to its favorable balance between recall accuracy, query latency, and memory usage.

HNSW has been deployed at scale in a broad range of production systems in the past few years—everything from billion-vector image-retrieval services at big e-commerce platforms to real-time recommendation engines in streaming media, to adding context to large language model queries [2–4, 7]. Its graph search approach not only beats tree and hashing-based ANN methods when recall is critical but also provides a strategy for graceful degradation as a query's volume fluctuates while ensuring tight latency SLAs. Despite these successes, there is no standard process for choosing HNSW hyperparameters to meet both throughput goals and memory limits. In practice, engineers usually rely on handcrafted heuristics or ad hoc grid search, both of which can be costly in time and vulnerable to miscalibration as the characteristics of the dataset change.

Existing work on HNSW predominantly adopts a "one knob at a time" evaluation approach or treat it as an optimization problem where they search the Pareto frontier for an optimal cartesian pair for related hyperparameters such as build hyperparameters, where each parameter is varied in a controlled environment to analyze their independent effects [1, 5, 6, 10]. While such an approach can accurately quantify individual parameter effects, it is a significant oversimplification of real-world scenarios where multiple parameters interact simultaneously, seemingly in non-linear and unpredictable ways.

Automated hyperparameter optimization frameworks can efficiently converge on high-performing HNSW settings with little work. However, the frameworks act as black-box optimizers. Practitioners rarely learn why a given combination of $efConstruction$, $efSearch$, and M yields certain trade-offs, nor how those trade-offs transfer across datasets of differing size and dimensionality. Our work systematically maps out how these three key parameters jointly impact recall, latency, build time, and memory across *multiple* benchmarks. By producing interpretable trade-off curves and lightweight analytical models, we enable practitioners to predict performance on unseen workloads, sanity-check autotuner outputs against quantitative baselines, and establish reproducible defaults. Thus, even in an era of powerful autotuners, our white-box, multi-parameter evaluation remains essential for transparent, cross-dataset insight and reliable production deployment.

To address these shortcomings, our work systematically explores a comprehensive multi-dimensional parameter space, covering:

- **Dataset Size** : spanning from 1 million to 1 billion vectors,
- **Dimensionality** : ranging from 10 to 1000 features,
- **Key HNSW parameters** : $efConstruction$, $efSearch$, and M.

We will evaluate the impact of these parameters on five metrics:

- **Recall** : to assess accuracy of retrieved neighbors
- **Query Latency** : for real time responsiveness
- **Throughput** : reflecting parallel efficiency
- **Index build time** : to quantify offline cost
- **Memory usage** : for capacity planning

We not only intend to conduct a benchmark but also to explore the possibility of deriving analytical models that capture the underlying trends in performance. These models could serve as lightweight predictors or or cost estimators. Through this study we aim to move beyond isolated empirical results to try and bridge the gap between benchmarking and theoretical modeling, making it better for both researchers and practitioners.

2 Related Work

We provide a systematic survey of existing works. We describe their limitations in comparison to our work.

2.1 Hierarchical Navigable Small Worlds

Malkov et al. provide a detailed and comprehensive analysis of HNSW [5]. However, their analysis is one-dimensional. They acknowledge that hyperparameter tuning is important but they do not analyze how setting multiple hyperparameters at once affects the system.

2.2 Automatically Tuning HNSW construction parameters

We mentioned why auto-tuning parameters will theoretically come short when it comes to the soft side of developer experience, but it's still important that they are discussed in contrast to our work explicitly. One work looked at automatically tuning the HNSW construction parameters using GridSearch specifically to tune the hyperparameters of $efConstruction$ and M so that the corresponding HNSW has the best recall. They only, however, look at recall [10]. Yang et al. proposed a learning-based approach to tune vector database management systems and recognize that hyperparameters are interdependent but their work focuses on recall and speed; they do not focus on other potentially also important metrics [9]. Another work examines exploring search-time settings for the optimal hyperparameter settings. They test the Cartesian products of the hyperparameters and apply pruning to shrink the search space. However, they only care about speed and accuracy as primary metrics [2].

2.3 ANN algorithm benchmarking

Aumuller et al. propose a tool for evaluating the performance of in-memory approximate nearest neighbor algorithms [1]. They note the build parameters for HNSW all blend seamlessly into each other but do not examine how the build and query parameters together can affect the metrics of HNSW.

3 Proposed Method

To systematically evaluate the performance of the HNSW (Hierarchical Navigable Small World) indexing algorithm, when tuning multiple hyperparameters at once, we designed and implemented a benchmarking pipeline using the FAISS library's `IndexHNSWFlat` class [2]. Our primary objective is to understand how different multi-hyperparameter configurations influence the trade-offs between accuracy, speed, and resource usage.

We focused on three critical HNSW parameters:

- **M**: the number of bi-directional edges per node,
- **efConstruction**: the size of the dynamic candidate list during index construction,
- **efSearch**: the candidate list size during query-time search.

To explore the configuration space, we defined a grid of values:

- $M \in \{4, 8, 16, 24, 32\}$
- $efConstruction \in \{50, 100, 200\}$
- $efSearch \in \{10, 50, 100, 200\}$

This results in a total of $5 \times 3 \times 4 = 60$ unique experiments per dataset. For each configuration, we:

1. Build the HNSW index on the training set,
2. Evaluate search performance using the test set and provided ground-truth neighbors,
3. Record key performance metrics, including:
   - Recall@1
   - Average query latency (ms)
   - Throughput (queries per second)
   - Index build time (seconds)
   - Memory usage (MB)

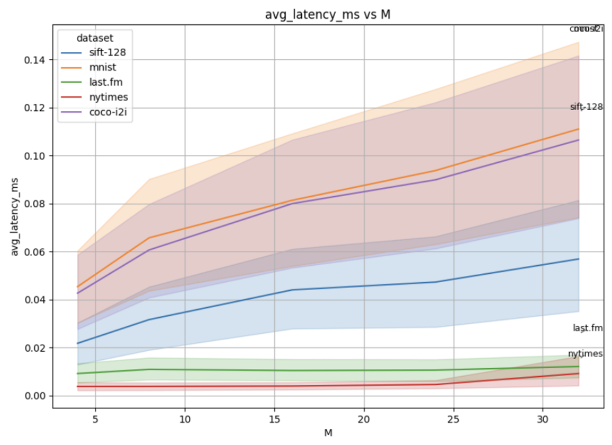

Figure 1: Line plot of the average latency per query versus varying values of M, over different datasets. Lower values are better.

The experiments were conducted across five datasets with varying sizes and dimensionalities from the ANN-Benchmarks suite, including SIFT (128-D, 1M train data), MNIST (784-D, 60K train data), Last.fm (64-D, 292K train data), NYTimes (256-D, 290K train data), and COCO image-to-image (512-D, 330K train data) [1]. This diversity in both feature dimension and dataset scale ensures that our evaluation framework captures real-world search scenarios and generalizes across use cases with different data characteristics.

By systematically evaluating all 60 configurations per dataset, we are able to capture both expected trends (e.g., increasing M typically improves recall) and more nuanced trade-offs (e.g., how $efSearch$ impacts latency more sharply than build time). All results are saved in structured CSV files, allowing for further aggregation and plotting, as detailed in the results section.

4 Results

We begin by evaluating the effect of individual hyperparameters on system performance. When plotting key metrics (such as recall, build time, latency, and memory usage) against a single parameter (e.g., M, efConstruction, efSearch), many of the results match our expectations. For instance, as shown in Figure 1, increasing M tends to improve recall due to better graph connectivity. However, this comes with trade-offs: higher M values lead to more edges in the graph, which increases both the time required to build the index and the latency during search. More valuable insights appear when we analyze how metrics vary across multiple hyperparameters.

4.1 Extracting Insights from Two-Parameter Performance Curves

In particular, as an example, we consider the plots of recall, latency, and build time across combinations of efSearch and M, and memory usage across both dimension and M.

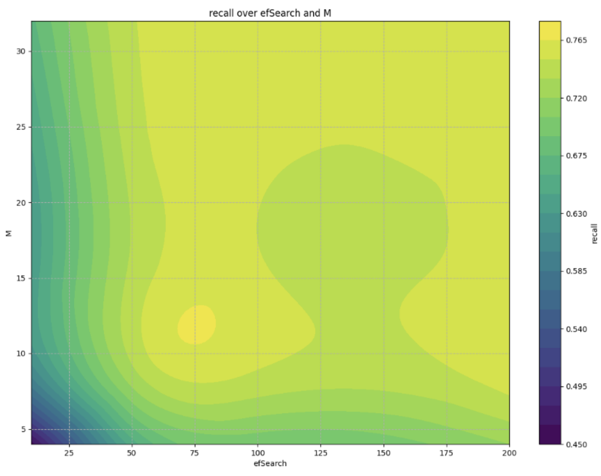

Figure 2: Contour plot of recall over varying values of $efSearch$ **and** M**, averaged across different datasets. Lighter colors are better.**

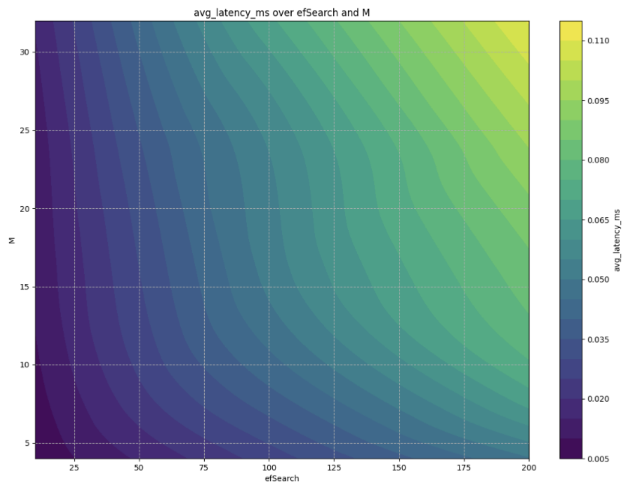

Figure 3: Contour plot of average latency per query over varying values of $efSearch$ **and** M**, averaged across different datasets. Darker colors are better.**

In Figure 2, we observe that recall increases when either $efSearch$ or M is increased. This makes sense: $efSearch$ controls how thoroughly the graph is searched at query time, while M controls the number of connections each node has, improving graph quality. Notably, both parameters contribute to recall improvements at similar rates. This raises a practical question: if a provider wants to increase recall, should they increase M, $efSearch$, or both?

To help answer that, we examine Figure 3, which plots average query latency over efSearch and M. Here we see a key distinction: while increasing either parameter raises latency, the effect is much more pronounced with efSearch. The latency grows quickly as efSearch increases, while changes in M have a relatively mild impact. This suggests that increasing M may be the more latency-efficient way to boost recall.

However, this comes with a different cost, shown in Figure 4. Build time increases significantly with larger M values, since more edges must be computed and added during index construction. On the other hand, increasing efSearch has almost no impact on build time, as it only affects the query-time behavior. This highlights a classic engineering tradeoff: choosing between faster index build time or better query-time performance. A provider that values quick setup (e.g., frequently updating indices) may prioritize tuning efSearch, whereas one optimizing for fast search may prefer to increase M instead.

Lastly, Figure 5 plots memory usage per data point as a function of both dimension and M. As expected, increasing dimension leads to a substantial rise in memory consumption, since each vector requires more space. Interestingly, increasing M has a relatively minor impact on memory usage in comparison. While M does introduce more edges into the graph structure, the memory overhead from these additional connections is small compared to the cost of storing higher-dimensional vectors. This suggests that providers concerned about memory constraints should pay closer attention to dimension, as it dominates memory footprint, while M can often

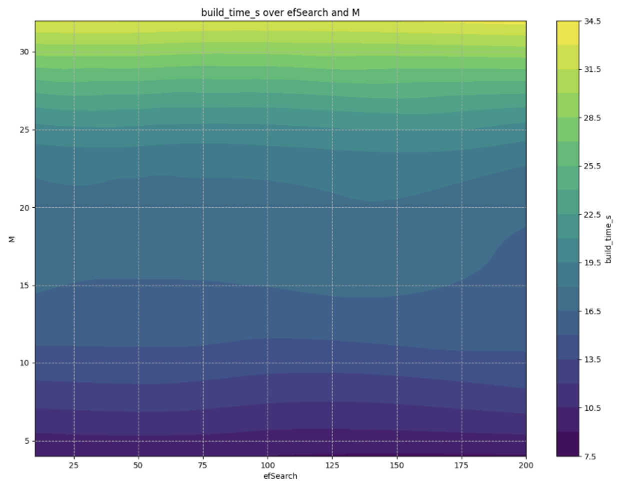

Figure 4: Contour plot of build time over varying values of $efSearch$ **and** M**, averaged across different datasets. Darker colors are better.**

be increased for performance gains without incurring a significant memory penalty.

Overall, these plots provide actionable guidance for different use cases. A provider prioritizing fast search performance might choose to increase M, accepting a longer build time and higher memory usage. Another provider who rebuilds indices frequently might lean on efSearch instead, avoiding costly builds while tolerating slightly slower queries. These multi-hyperparameter visualizations make the tradeoffs between accuracy, speed, and resource usage much clearer, helping system designers align parameter choices with user needs and infrastructure limitations.

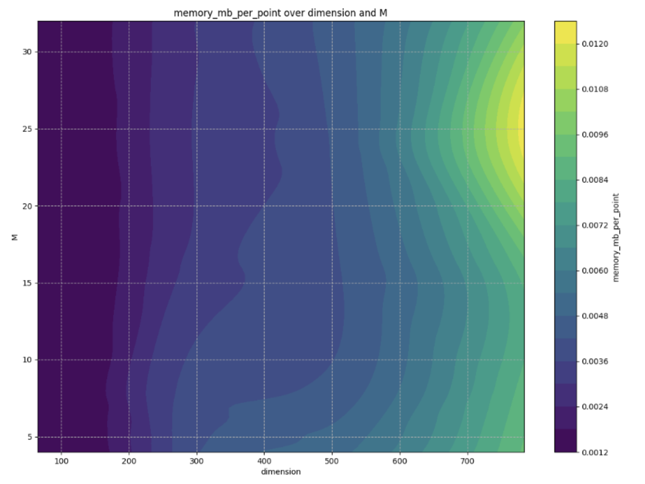

Figure 5: Contour plot of memory over varying values of *dimension* and *M*. Darker colors are better.

All additional plots, including variations across different hyperparameters and metrics not shown here, are available in our GitHub repository (https://github.com/Rachanon-Andrew-Petchoo/HNSW-evaluation) under the `plots` folder for further reference and exploration.

4.2 Effect of Three Hyperparameters on Performance Metrics

Figure 6, Figure 7 plot the three hyperparameters of $efSearch$, $efConstruction$ and M with color shading representing the median average latency (milliseconds), recall respectively.

We observe that the latency increases as we increase $efSearch$ consistently regardless of what $efConstruction$ and M are set to. This is not surprising as $efSearch$ influences the speed of searches in HNSW [5]. Additionally, we observe when $efSearch$ is set to highest parameter, increasing M leads to an increase in latency across all configurations which is also in line what we expect to see since the M hyperparameter also influences the query speed. Overall we note that these findings are not surprising at all.

We observe for recall in general an increase in recall as increase $efSearch$ with two distinct clusters located around an $efConstruction$ of 50 and $efSearch$ of 200 as well as an $efConstruction$ of 200 and an M of 24. The cluster that is closer to an $efConstruction$ of 100 is not surprising as it was shown that an $efConstruction$ value of 100 is ideal for most cases [5]. Interestingly, continuing to increase the $efConstruction$ parameter provides better recall when the M parameter is also raised at the same time, contrary to the fact. analysis done by the original authors. We observe that increasing M does in fact produce better recall, but not consistently across when the other two parameters are fixed. When $efSearch$ is smaller, increasing M does not always lead to an increase in recall.

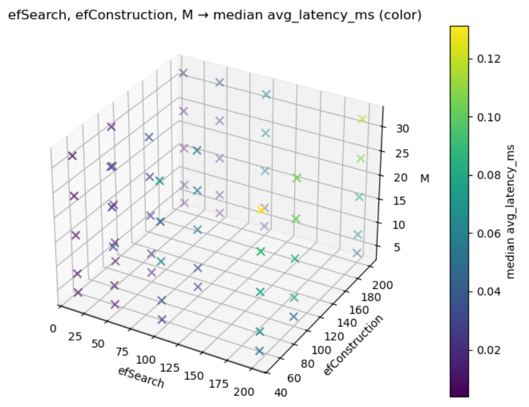

Figure 6: Scatter plot of the three hyperparameters of efSearch, efConstruction and M. The color shading represents the median average latency in milliseconds, where darker colors are better.

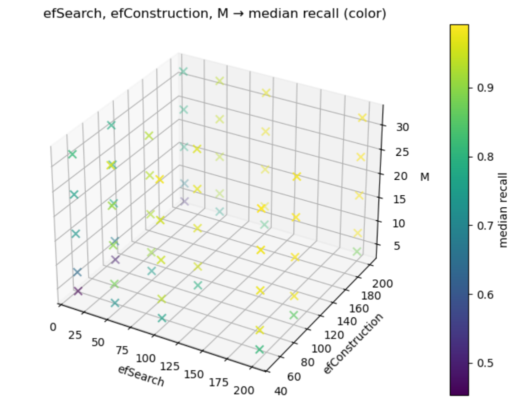

Figure 7: Scatter plot of the three hyperparameters of efSearch, efConstruction and M. The color shading represents the median recall in milliseconds, where lighter colors are better.

References

[1] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2018. ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms. arXiv:1807.05614 [cs.IR] https://arxiv.org/abs/1807.05614

[2] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The faiss library. *arXiv preprint arXiv:2401.08281* (2024).

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

[4] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems* 33 (2020), 9459–9474.

[5] Yu. A. Malkov and D. A. Yashunin. 2018. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. arXiv:1603.09320 [cs.DS] https://arxiv.org/abs/1603.09320

[6] Philip Sun, Ruiqi Guo, and Sanjiv Kumar. 2023. Automating nearest neighbor search configuration with constrained optimization. *arXiv preprint arXiv:2301.01702* (2023).

[7] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. 2021. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*. 2614–2627.

[8] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *arXiv preprint arXiv:2101.12631* (2021).

[9] Tiannuo Yang, Wen Hu, Wangqi Peng, Yusen Li, Jianguo Li, Gang Wang, and Xiaoguang Liu. 2024. Vdtuner: Automated performance tuning for vector data management systems. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 4357–4369.

[10] Wenyang Zhou, Yuzhi Jiang, Yingfan Liu, Xiaotian Qiao, Hui Zhang, Hui Li, and Jiangtao Cui. [n. d.]. Auto-Tuning the Construction Parameters of Hierarchical Navigable Small World Graphs. *Available at SSRN 4925468* ([n. d.]).