National & Kapodistrian University of Athens
School of Science, Department of Informatics and Telecommunications
Athens, January 2025

Project Report: ANN with Vamana Indexing

Bekos Gerasimos (sdi2100113)
Porichis Dimitrios Stefanos (sdi2100159)

Abstract

The implementation and optimization of Approximate Nearest Neighbors (ANN) search algorithms are critical for efficient high-dimensional data retrieval in modern information systems. Throughout this semester, we developed and optimized multiple Vamana-based ANN implementations as part of the "Software Development for Information Systems K23a" course at NKUA. This report presents relevant conclusions derived from our experiments with thread parallelism and other optimization tactics.

Keywords: ANN, Vamana, Stitched, Filtered, Parallelism, Randomness

1. Introduction

Vamana indexing is a powerful and efficient method for approximate nearest-neighbor searches, widely recognized for its scalability and adaptability in handling large-scale datasets. Unlike traditional brute-force KNN approaches, which involve comparing a query point to every data point, Vamana employs graph-based structures to significantly reduce the number of comparisons required.

This project focuses on the implementation, optimization, and comparative evaluation of ANN searches using Vamana-based indexing strategies. Specifically, we explore different versions of Vamana, including stitched and filtered variants, trying to incorporate techniques such as thread parallelism and caching to reduce computation time.

By examining the trade-offs between accuracy, computational complexity, and memory usage, our study provides practical observations on the performance of these indexing techniques in specific scenarios.

2. Relevant background for Vamana Versions

Vamana-based implementations work in two phases; Graph creation and Query Making.

The first stage creates a graph representation, with nodes being instances of our dataset

and edges representing prospective neighbor relationships based on distance metrics. This graph needs to be structured in such a way that when starting from a starting point s, we can efficiently navigate through the graph to find the closest neighbors of a query point q.

The second stage utilizes the graph created to search for neighbors of arbitrary points. It does so by performing a greedy search, starting from a point s and following the best candidates relative to the query point q.

Vamana versions differ in the ways they create and search on those graphs. Some, like Filtered-Vamana and Stitched-Vamana, even support filtered searches – populating results that include only specific categories of nodes.

Below, we present a brief overview of how each of the three versions of Vamana works to help readers better understand our suggested optimizations.

Please note that this paper will not discuss the reasoning behind these algorithms, as they diverge from the research topic. Readers who want to further understand such decisions can refer to the linked sources: [Subramanya et al. (2019)], [Gollapudi et al. (2023)]

2.1 Classic Vamana

Classic Vamana poses the basis for all versions, and it is suitable for performing unfiltered ANN searches.

The Vamana index is constructed using an initial random, well-connected graph, which is later refined by performing a *GreedySearch* upon all points to find their best neighbor candidates. Each neighborhood proposal is evaluated by a *RobustPrune* function, which settles the neighbors of the node. Finally, mirrored mutual connections are suggested from all the resulting neighbors, before proceeding to the next iteration.

To perform queries, a *GreedySearch* starts from a given node s, moving towards neighbors that are closer to its target node q, keeping track of its best options along the way.

For this search to work, both in indexing and query-making, a good starting point, preferably near the center of the graph, must be selected. To do so, Vamana calculates the medoid of the graph at the start of its creation, a computationally intense procedure.

2.2 Filtered Vamana

Filtered Vamana introduces support for performing category-specific queries. Each node, apart from its coordinates also has a category parameter –referred to as its filter.

The graph creation follows the same form as Classic Vamana with some key differentiating factors. For one, the graph is not initialized to a random state, resulting in a more sparse graph. Secondly, the greedy and pruning methods are tuned to prioritize, or exclusively support, same-category connections, resulting in graphs that are heavily connected on a per-filter basis.

The starting point for the searches needs to be category-dependent, with each filter having its own medoid. To minimize computational overhead, filtered Vamana opts for a random medoid assessment, picking one arbitrary point out of each sub-set.

Please note that in its original format, Filtered Vamana supports nodes being in multiple categories. Our study focuses on the scenario where **each node can belong to exactly one category**.

2.3 Stitched Vamana

Stitched Vamana implements filtered-enabled graphs in a slightly different way than Filtered-Vamana. Instead of performing the graph generation per node, it simply breaks down the dataset into subsets with the same category and performs a classic Vamana initialization to each of them. That being the case, medoid calculations are done using brute force for each sub-category.

Its core implementation supports multi-category points, but as stated before we will only look at single-category points.

3. Suggested Optimizations

After implementing the above Vamana versions as part of our 2 first assignments, we moved on to the optimization phase. Our main focus where to improve execution times, and query recall rate.

For the execution time, we searched for high-intensity, non-dependant calculations that could be parallelized, possible code refactoring, and caching of repetitive calculations. As for the query recall rate, we experimented with inserting randomness at the initialization phases of filtered graphs.

Below we present the attempts made to each Vamana version.

3.1 Classic Vamana

With our core implementation being already accurate enough, our main efforts went towards finding ways to decrease construction times. As the iterations of the refining loop have data dependencies, we attempted to lower the time needed to calculate the medoid point, which acts like a starting point for all searches. We went over 4 different calculation ways:

- **Brute Force Medoid**: Our initial implementation.
- **Parallel Brute Force Medoid**: Calculates the true medoid, splitting the points workload to n threads.
- **Random Subset Medoid**: Calculates the true medoid of a randomly chosen subset of data.

- **Random Medoid**: Selects a random point from the whole dataset.

3.2 Filtered Vamana

Given that our dataset included only data points with up to one category, previous Filtered Vamana implementations were horrendous when performing unfiltered searches, as they created smaller unconnected sub-graphs. In our initial implementation, we conducted a collective greedy search of all categories and kept the k closest of those, but the numbers were not outstanding either. Taking things a step further, we introduced random initialization at the graph creation, starting from a well-connected graph as in Classic Vamana.

To decrease the graph creation times, we implemented parallelism on a per-category basis, as concurrently updating different category nodes won't result in dependency issues. Note, that this can not be done when random initialization is used or the dataset contains multi-category points unless further synchronization is added.

3.3 Stitched Vamana

Stitched Vamana posed the same deficiencies as Filtered Vamana, the only difference being that its sub-graphs were better connected, due to being Vamana graphs. As a result, we followed a similar course of action, parallelizing categories and introducing random initialization.

Additionally, we redesigned our initial Stitched Vamana Graph from being a map with subgraphs to a unified graph that can be later processed as per-category smaller ones.

3.4 General Optimizations

3.4.1 Caching

We implemented an LRU cache for storing distance data, in hopes it will reduce calculation repetition when performing mutual neighbor relations and robust pruning after greedy searches.

3.4.2 Query Making Parallelism

As the graph is already created, and queries do not alter its form, we suggest query-making parallelism as a way to increase result throughput. This can be done by splitting the workload into n independent threads and evaluating their partial results at the end.

3.4.3 Refactoring

General refactoring was done to the entirety of our project, further decreasing our execution times. The most notable of those were:

- **Moving to a single-category specific implementation**: As each point can belong only to one filter category, we changed our implementations to better include only a category flag at each point rather than a category set.

- **Using squared Euclidean distance**: Shifting away from Euclidean distance to just a sum of the differences risen to the power of 2 greatly decreased our execution time while not altering the results.
- **Utilizing compile-time optimization**: Our project was compiled using O3 and the ffast-math g++ flags

3.4.4 Data types

To create a well-performing implementation, we used data structures that match the behavior of each component. To name a few:

- A graph consists of a vector for storing nodes, a set for storing category types, a map for fast per category medoid retrieval, and other metadata fields.
- Nodes consist of a float array containing their coordinates, a set for storing their neighbor relations, and other metadata fields
- Neighbor relations are represented with a pointer to the prospective neighbor, as well as the distance between them, minimizing the need for recalculations.
- All set calculations needed for Greedy Search and Prunning utilize special sets that order neighbor candidates based on their distance, minimizing min calculations.

4. Experiments Structure

To assess the performance gains of each implementation, we conducted experiments in a variety of data set sizes (10k, 20k, 50k, and 100k points), keeping track of three metrics: time, space needed, and recall rate.

All experiments were run multiple times to mitigate background activity noise and factor in luck in implementations using randomness. *(2 to 8 times depending on the experiment size).*

4.1 Test-bench specifications

The machine used for conducting the experiments was a DELL laptop with the following specifications:

- **CPU:** Intel® Core™ i7-10750H (6-Cores)
- **RAM:** 8GB
- **OS:** Ubuntu 24.04.1 LTS
- **C++ Compiler**: g++ (Ubuntu 13.3.0-6ubuntu2 24.04) 13.3.0

Due to the low specifications of our machine, experiments that require more than 5GB of RAM or >6 hours execution time will be considered DNF.

4.2 Execution Parameters

We used the same execution parameters for all experiments.

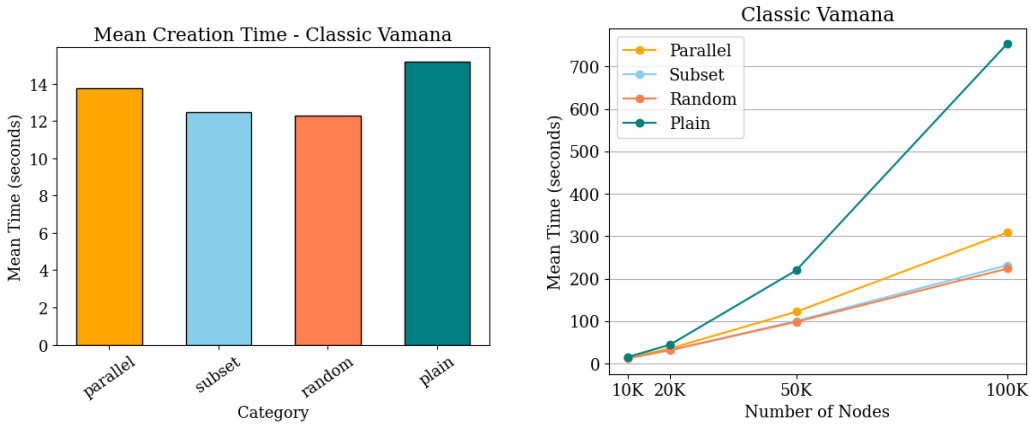
- **For Classic Vamana:** $R = 20$, $L = 75$, $a = 1.2$
- **For Filtered Vamana:** $R = 96$, $L = 90$, $a = 1.2$
- **For Stitched Vamana:** $R = 32$, $R_s = 64$, $L = 100$, $a = 1.2$

[Results start on the next page for a better layout]

5. Results

5.1 Classic Vamana Medoid Optimization

Conducting our experiments for the proposed medoid calculation alternatives [Section 3.1], we found that inserting randomness in the selection of the medoid greatly decreased the execution times [Figure 1], while having no impact on the overall accuracy of queries [Figure 2.a].

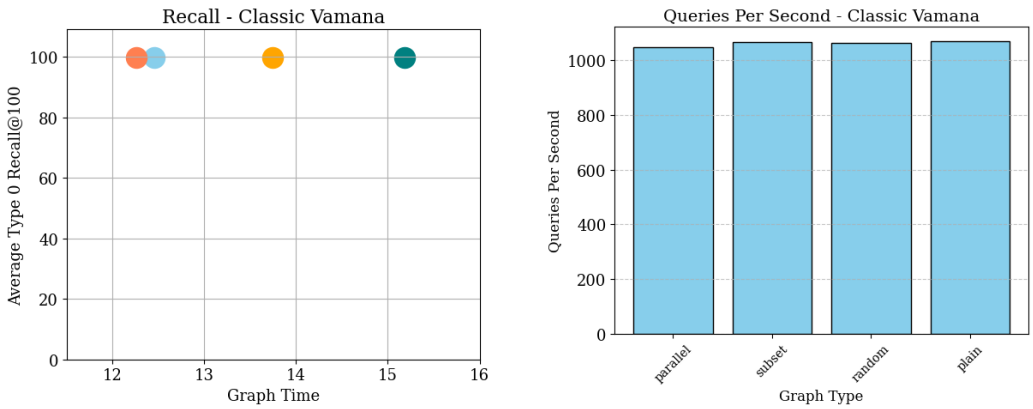


(a) 10K Dataset

(b) Increasing Data Set size

Figure 1: Classic Vamana Index Creation

Purely Random and Random Subset also scale better over bigger data, as they are not tied to the dataset's size. Parallel medoid also performs substantially better than the original implementation but gets slower in growing data numbers.



(a) 10k dataset accuracy

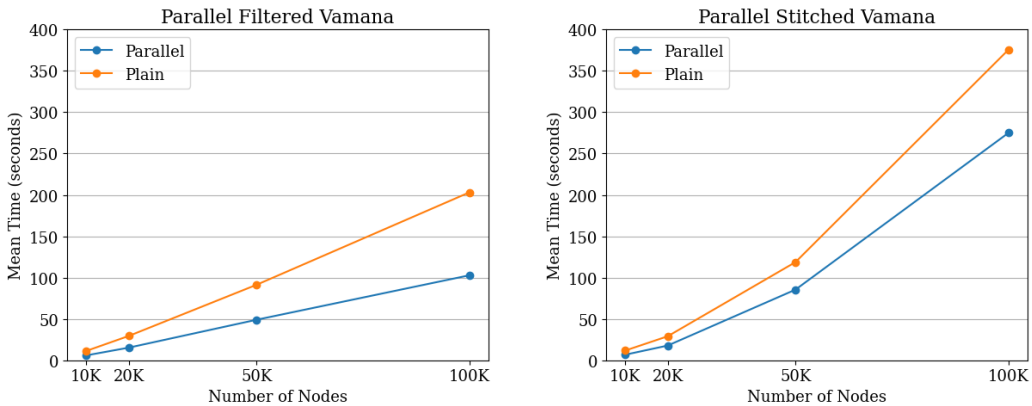
(b) Query making speed

Figure 2: Query Accuracy and Speed in 10k data set

No substantial changes were found in the time needed for query responses, with all implementations averaging 1020 queries per second. [Figure 2.b]

5.2 Parallel Index Creation

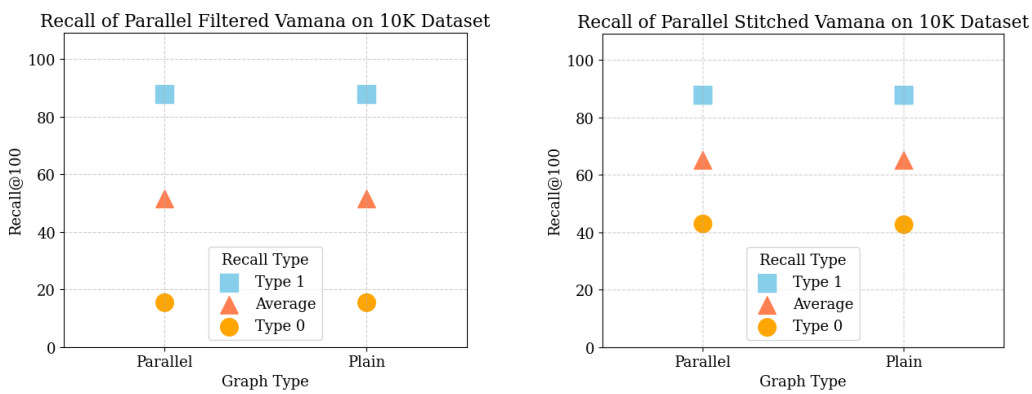
The index creation of Stitched and Filtered Vamana Implementations poses a great opportunity for parallelism, as each iteration can be divided to execute concurrently based on the categories. [Sections 3.2, 3.3]



(a) Filtered Vamana (b) Stitched Vamana

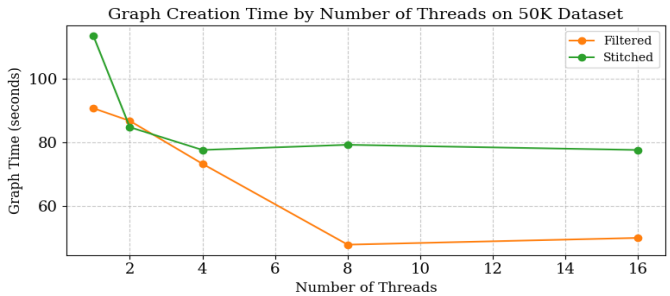
Figure 3: Index Creation Times using Parallelism (8-threads)

As the data suggest, utilizing parallelism decreases creation times noticeably, with the stable accuracy between the implementation validating that no dependencies messed with the data.



(a) Filtered Vamana Accuracy (b) Stitched Vamana Accuracy

Figure 4: Query Accuracy in 10k dataset



Threads	Stitched	Filtered
1	113.41	90.71
2	84.76	86.75
4	77.60	73.25
8	79.22	47.81
16	77.59	49.92

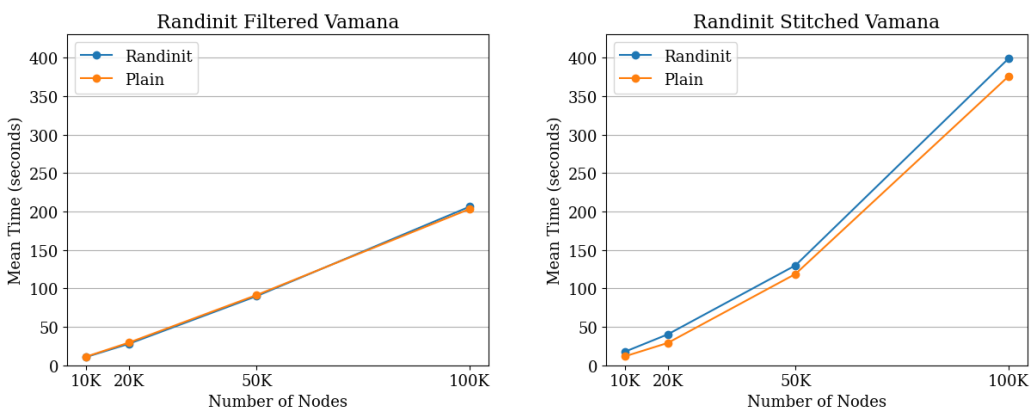
Figure 5: Index creation times in a growing number of threads

By increasing the number of threads, as shown in [Figure 5], we see execution times declining drastically for Filtered Vamana, stabilizing when using 8 threads or more, while Stitched Vamana follows the same pattern with a visible overhead.

The main reason for that latency is the medoid calculations done for each Vamana subgraph Stitched creates, as it uses the original brute force method.

A potential solution to this issue could be using the Random Medoid Calculation suggested in 3.1. Sadly, this report **will not** conduct such experiments.

5.3 Random Initialization



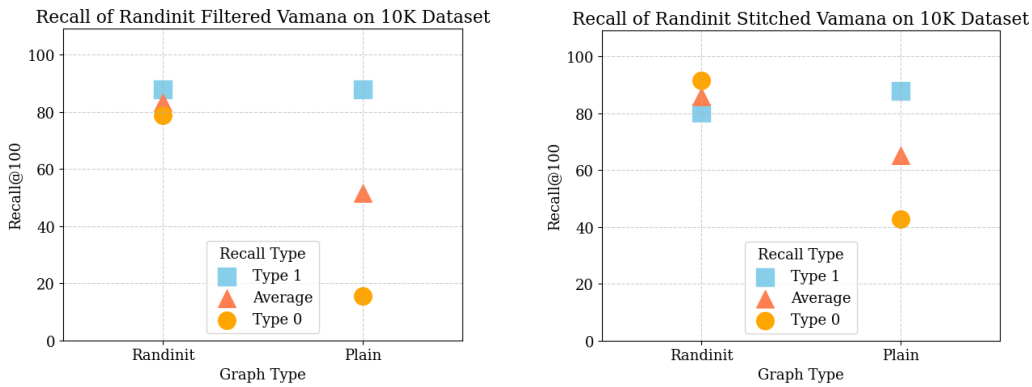
(a) Filtered Vamana

(b) Stitched Vamana

Figure 6: Index Creation Times using Randinit

Filtered and Stitched Vamana's low performance on unfiltered queries is an evident problem when using single-category datasets [Sections 3.2, 3.3]. Our results suggest adding

random connections at the initialization phase, helps mitigate this issue, with small potential hits in the accuracy of filtered searches [Figures 7.a, 7.b].

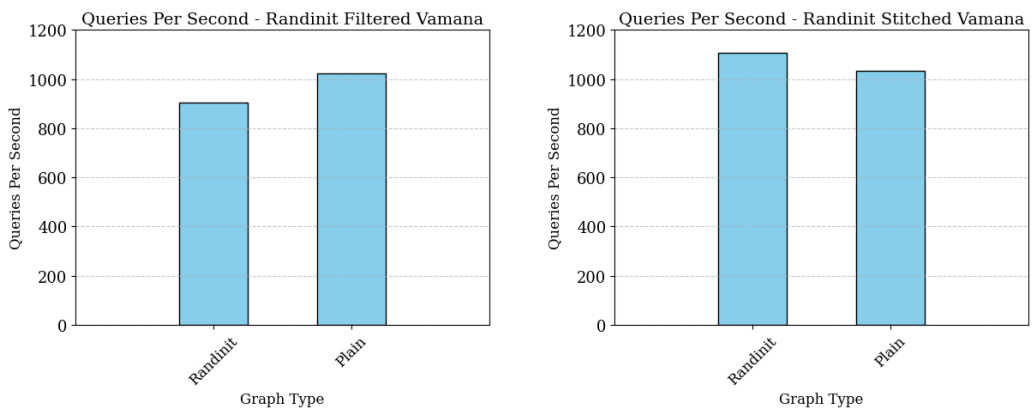


(a) Filtered Vamana

(b) Stitched Vamana

Figure 7: Query Accuracy in 10k data set

Computationally both implementations seem to perform similarly, as far as graph creation is concerned [Figure 6]. Query-making speeds, on the other hand, present shifts when randomness is used, being faster in Stitched Vamana [Figure 8.b] and slower in Filtered Vamana [Figure 8.a].



(a) Filtered Vamana

(b) Stitched Vamana

Figure 8: Query speeds differences using randinit

Derived from the above random initialization can be suggested as a way to find the balance between different query types accuracy, which could be useful in some applications.

5.4 Caching

As finding distances is computationally expensive, our main goal with implementing an LRU cache was to minimize recalculations of the same pairs [Section 3.4.1]. On the contrary, our data suggest that such a cache does not produce enough cache hits to compensate for its updating and memory costs.

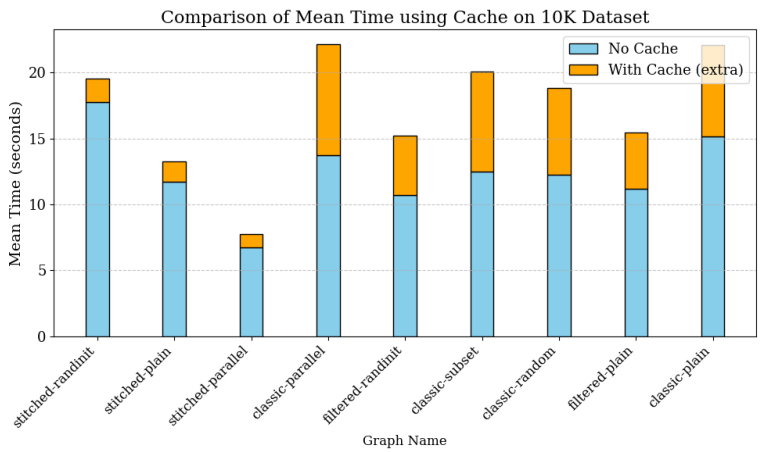


Figure 9: Index creation times in 10k dataset with cache

As seen in the figures, all graph creation procedures with cache enabled perform worse than their cache-less counterparts, even when data sizes increase.

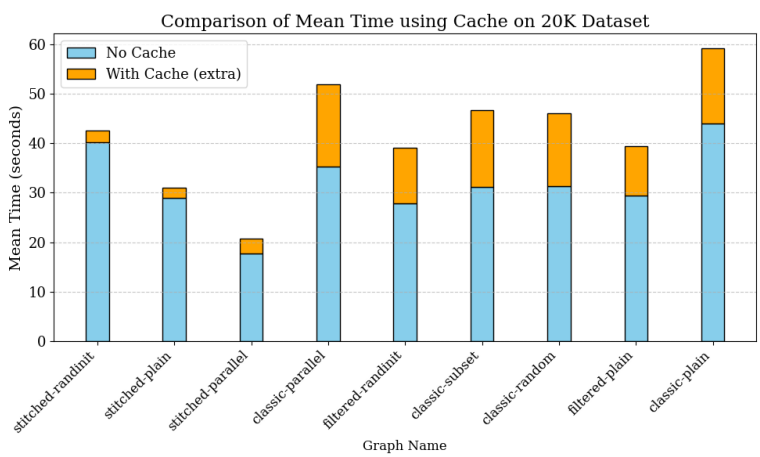


Figure 10: Index creation times in 20k dataset with cache

This outcome could be due to our neighbor struct's incorporation of distance which reduces the number of redundant calculations, or LRU not being the right replacement strategy.

5.5 Parallel queries

Queries are performed in already created, stable graphs by using only read operations on them. As a result, no data dependencies are present, making them ideal for parallelization. [Section 3.4.2]

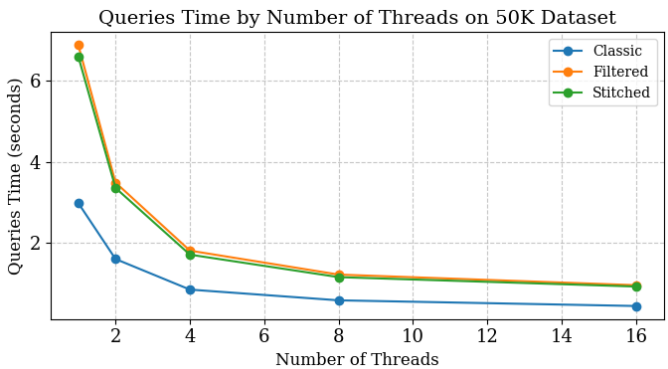


Figure 11: Execution times of queries with different thread counts

Our data proves this claim, showcasing the astonishing decrease in time needed to perform the same number of queries when using more threads.

For reference: Classic performs 2516 queries in all executions, while Stitched and Filtered perform 5012 (2516 unfiltered and 2496 filtered).

5.6 Space complexity

Finishing our experiments, we documented the space complexity of our different implementations to better understand the trade-offs of increasing performance. The metric used is peak simultaneous memory usage.

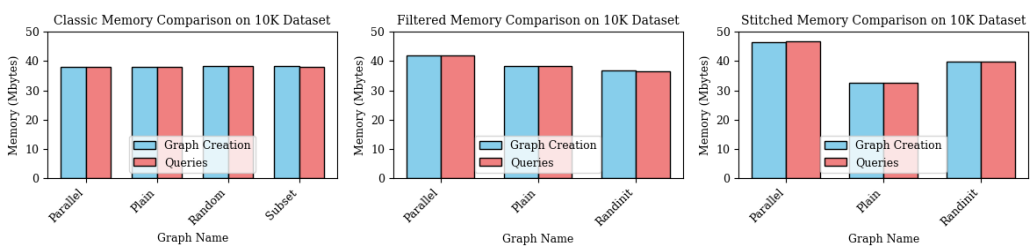


Figure 12: Memory Usage for 10k dataset

Evidently, parallel implementations require more memory to run than their serial competitors, as they compute based on more data concurrently. For machines low on RAM storage, this could become a potential issue, especially when datasets get larger.
Randinit variants also require more storage, mainly because of the increase in neighbor counts from the early stages.

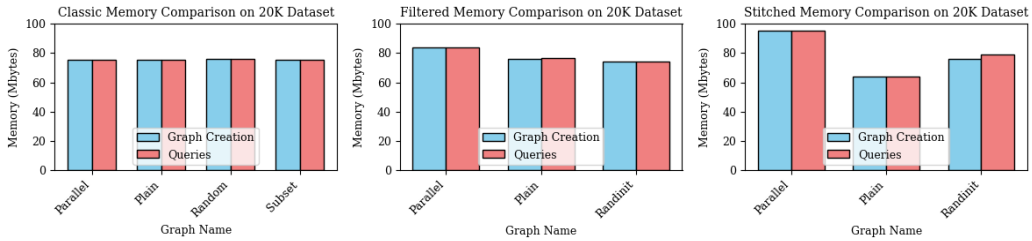


Figure 13: Memory Usage for 20k dataset

On the contrary, all changes in Classic Vamana implementations do not have an impact on peak memory usage, as calculating medoids is not the most memory-intensive task.

5.7 Experimenting with Huge Datasets

Due to the low specifications of our test bench, experimenting with huge datasets was not sustainable or reliable. Just to provide some context on how implementations scale, we provide the following table describing trial executions on a 1-million-point dataset.

Implementation	Index Creation Time (100k)	Index Creation Time (1m)
Plain Classic Vamana	12 minutes 33 seconds	No reliable data
Parallel Classic Vamana	5 minutes 8 seconds	2 hours 40 minutes
Subset Classic Vamana	3 minutes 51 seconds	1 hour 8 minutes
Random Classic Vamana	3 minutes 43 seconds	43 minutes
Randinit Filtered Vamana	3 minutes 26 seconds	40 minutes
Plain Filtered Vamana	3 minutes 23 seconds	36 minutes
Parallel Filtered Vamana	1 minute 42 seconds	18 minutes
Plain Stitched Vamana	6 minutes 15 seconds	No reliable data
Randinit Stitched Vamana	6 minutes 38 seconds	No reliable data
Parallel Stitched Vamana	4 minutes 35 seconds	DNF (Memory Maxed)

Table 1: Trial executions with huge data

6. Conclusions

Throughout our experiments, we derived useful information regarding possible optimizations of different Vamana Versions, as well as identifying scenarios that make sense to use them.

For one, we have seen that Classic Vamana does not drastically depend on the accuracy of the medoid passed as a starting point. This creates room for faster implementations

that utilize statistical randomness to select one, reducing computational burdens.

Additionally, we found that the addition of randomness in Filtered Vamana and Stitched Vamana index creation can help drive up accuracy on unfiltered searches, potentially creating graphs optimized for balanced workloads of filtered and unfiltered searches.

Introducing parallelism in the graph creation proved to be a good way to create indexes faster, leveraging the multi-threading capabilities of modern CPUs, with the only trade-off being high concurrent memory usage.

Finally performing queries concurrently, turned out to be the optimal way to increase result throughput.

7. Further Research Ideas

Based on our results, parallelism, and random initialization pose great opportunities for Filtered and Stitched Vamana implementations.

Further experimentation can be done to introduce concurrent index creation in multi-category points as well as in randomly initialized graphs.

Finally, we believe experimenting with random medoid calculations in Stitched Vamana is another topic worthy of research.

8. Repository

All the code is available in our Github repository.

Detailed execution logs from our experiments can be found in the project_results.xlsx file located in the same repository.

References

Gollapudi, S., Karia, N., Sivashankar, V., Krishnaswamy, R., Begwani, N., Raz, S., Lin, Y., Zhang, Y., Mahapatro, N., Srinivasan, P., Singh, A., and Simhadri, H. V. (2023). Filtered-diskann: Graph algorithms for approximate nearest neighbor search with filters. In *Proceedings of the ACM Web Conference 2023*, WWW '23, page 3406–3416, New York, NY, USA. Association for Computing Machinery.

Subramanya, S. J., Devvrit, Kadekodi, R., Krishaswamy, R., and Simhadri, H. V. (2019). *DiskANN: fast accurate billion-point nearest neighbor search on a single node*. Curran Associates Inc., Red Hook, NY, USA.