Vamana Indexing Performance Analysis

Stefanos Gkikas[1], Theodoris Mallios[1]

[1] *Kapodistrian University of Athens*

Abstract

This study presents a comprehensive technical analysis of implementing the Vamana algorithm family in C++. The Vamana Indexing algorithms, introduced in the seminal 2019 DiskANN paper, represent a state-of-the-art approach for constructing approximate k-nearest neighbor (k-NN) graph indices in database systems, offering an optimal balance between search speed and accuracy. Subsequent developments, notably Filtered Vamana and Stitched Vamana, enhanced the original algorithm by incorporating dataset filtering capabilities to enable constraint-based nearest neighbor approximations. Our investigation examines the fundamental technical challenges and implementation considerations encountered during the development of these three algorithmic variants.

1 Vamana

In our implementation of the Vamana algorithm's graph structure, we opted for an adjacency list representation over an adjacency matrix due to the dynamic nature of edge operations and memory efficiency considerations. The memory requirements of an adjacency matrix, scaling quadratically with the number of nodes ($N \times N$), would impose significant constraints on the algorithm's applicability to large-scale datasets. Our implementation utilizes a vector of unordered sets as the underlying data structure for the adjacency list, enabling constant-time complexity $O(1)$ for critical operations, including edge insertion, neighbor enumeration, and edge removal through hash table operations.

The Vamana algorithm incorporates three crucial hyperparameters: α, L, and R. The parameter α controls the balance between exploration and exploitation during the search process, influencing the algorithm's ability to navigate the graph efficiently. The parameter L represents the list size during the search, affecting the number of candidates considered at each step and, consequently, the algorithm's accuracy and runtime. Lastly, R defines the maximum degree of each vertex, imposing an upper limit on the number of outgoing edges, which is essential for maintaining graph sparsity and optimizing search performance.

Our implementation integrates two essential algorithmic components introduced in the DiskANN paper: `greedySearch` and `robustPrune`. The `greedySearch` algorithm computes the k-nearest neighbors of a given node from a candidate set of potential nearest neighbors. Our implementation employs a set data structure for maintaining the potential Nearest Neighbors Set (NNS) and an unordered set for tracking visited nodes. This design choice allows for logarithmic time complexity $O(\log n)$ for both insertion and pruning operations in the NNS. A key advantage of utilizing the set data structure is its ordered storage property, enabling constant-time $O(1)$ access to the closest point while maintaining efficient insertion and deletion operations. The visited nodes container, implemented as an unordered set, leverages its hash table structure to achieve constant-time $O(1)$ complexity for node visitation verification.

For the `robustPrune` function, we selected `std::set` (implemented as a Red-Black Tree) as the data structure for the candidate set, maintaining the ability to access the closest point in $O(1)$ time while preserving efficient insertion and deletion operations. To establish the distance-based ordering within these sets in both `robustPrune` and `greedySearch`,

Email addresses: stegiks@gmail.com, theodoris.mallios@gmail.com

we implemented a custom `CompareVector` struct that serves as the comparison operator, ensuring nodes are maintained in ascending order based on their distance from a reference point.

The complete Vamana algorithm implementation uses the aforementioned `greedySearch` and `robustPrune` components within an iterative framework that processes vertices in a randomized order to construct the approximate nearest neighbor graph. For each vertex, the algorithm executes `greedySearch` to identify potential neighbors, followed by a transformation of the visited vertices set from an unordered set to an ordered set for compatibility with the `robustPrune` function. The algorithm then applies `robustPrune` to refine the neighbor selection, ensuring graph quality while maintaining efficiency.

A critical aspect of the algorithm is the maintenance of the R-Regular constraint, where R represents the maximum allowed out-degree for any vertex in the graph. After the pruning phase, the algorithm evaluates each candidate edge against this constraint, only incorporating edges that maintain the R-Regular property. This bound on the maximum degree is essential for controlling the graph's density and ensuring efficient search operations.

1.1 Medoid Optimization in Vamana Algorithm

Performance analysis of the Vamana algorithm revealed that a significant computational overhead is attributed to the calculation of pairwise distances between points for determining the true medoid. We hypothesized that the algorithm's performance characteristics could be maintained using an approximated medoid selection approach rather than computing the true medoid. To validate this hypothesis, we conducted a series of experimental evaluations wherein the medoid was selected randomly with uniform probability from the point set. The comparative analysis focused on four key performance metrics: **Memory Usage**, **Recall Percentage**, **Graph Construction Time**, and **Query Speed**. Query performance was quantified in queries per second, while graph construction time was measured in absolute seconds.

Our experimental methodology encompassed a comprehensive parameter space exploration across multiple hyperparameters: $\alpha \in \{1.0, 1.1, 1.2\}$, $R \in \{32, 64, 96, 128\}$, and $L \in \{100, 135, 170\}$. The empirical results demonstrated that query performance remained statistically invariant under the random medoid selection strategy. However, the graph construction phase exhibited substantial acceleration due to the elimination of exhaustive pairwise distance computations, as illustrated in Figure 1. Notably, the impact on recall performance was negligible, with observed degradation not exceeding 0.2% – a variation that falls within the expected statistical deviation of random trials. Memory consumption remained constant between the optimized and baseline implementations, as the modified medoid selection strategy introduced no additional space complexity.

The experimental outcomes strongly support the adoption of random medoid selection as a viable optimization strategy, offering substantial improvements in index construction efficiency while maintaining the algorithm's effectiveness in terms of recall and query performance.

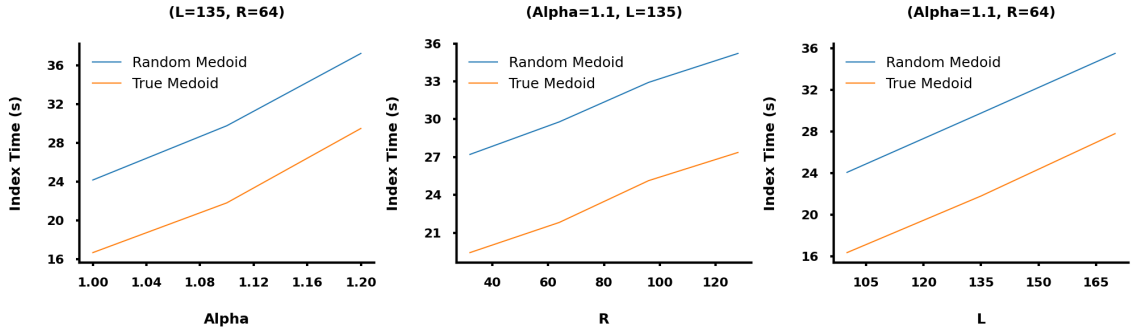

Figure 1. Comparative analysis of Vamana Index Graph construction time as a function of hyperparameters α, R, and L. The results demonstrate consistent acceleration across all parameter configurations when employing random medoid selection compared to the traditional true medoid computation method.

1.2 Parallel Implementation

In our analysis of parallelization opportunities within the Vamana algorithm, we encountered significant challenges due to the algorithm's inherent dependencies in edge manipulation operations. Initial attempts to implement parallel execution through Critical Sections during graph modifications proved ineffective, as the computational overhead introduced by thread synchronization negated potential performance benefits. Subsequently, we redirected our optimization efforts toward parallelizing distance calculations between points. Our implementation specifically focused on point-wise parallelization rather than dimensional parallelization, a strategic decision informed by the dimensional characteristics of our datasets (128 dimensions) relative to their cardinality (exceeding 10,000 points).

To evaluate the efficacy of this optimization approach, we conducted comprehensive experimental analyses to quantify the impact of parallel distance calculations on the overall Index Building Time. The results demonstrated that while the implementation incurred a modest increase in memory utilization, it yielded substantial improvements in computational efficiency. Notably, critical performance metrics, including Total Recall during evaluation and Query Speed, remained unaffected by the parallelization strategy. These findings validate the optimization's viability and suggest its potential applicability in alternative implementations of the algorithm.

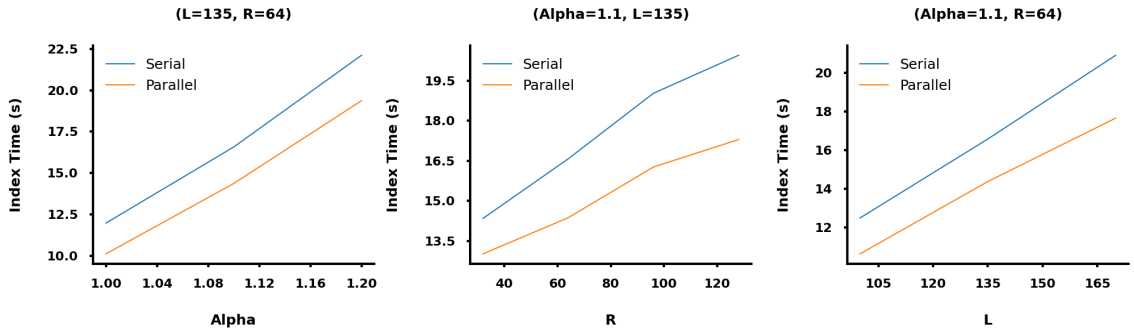

Figure 2. Index Building Time of Vamana algorithm in different hyperparameter settings. As is evident the Parallelization of distance calculation provides a major speedup in index building time.

2 Filtered Vamana

The Filtered Vamana algorithm is an extension of the basic Vamana implementation that adds filtered nearest neighbor search. This extension will create separate graph components for different filter values that are kept connected using a set of carefully chosen bridge edges. The algorithm first groups the dataset according to their filter values, such that nodes with the same filter value form a single group.

A critical step of initialization is the determination of the medoids for each filter, which will form the entry points for their respective subgraphs. This implementation now changes the filter to node mapping into a vector structure. In order to avoid systematic biases in graph construction, nodes are iterated in a stochastic manner.

To support the use of filters within the algorithms, we introduced unordered maps. One map is used to retrieve all vertices in the graph associated with a specific filter, while another map provides the medoid point for each filter.

With respect to each node processed, the algorithm carries out the following steps:

(1) Initializes a nearest neighbor set (NNS) utilizing a custom comparison operator based on vector distances
(2) Commences the search process from the filter group's designated medoid
(3) Executes `filteredGreedySearch`, a modified version of the original `greedySearch` that enforces filter constraints while identifying potential neighbors
(4) Converts the visited nodes set into an ordered structure for `robustPrune` compatibility
(5) Applies `robustPrune` to optimize the local graph connectivity while maintaining filter constraints

The implementation employs several key parameters to control graph construction:

- An `alpha` parameter governing search aggressiveness during construction
- An `L` parameter defining the beam width for the greedy search phase
- An `R` parameter constraining the maximum out-degree of nodes

One salient aspect of this method is the edge removal policy. Once the edges of a node are computed using `robustPrune`, the algorithm checks for mutual edges. For every neighbor j of the current node, the algorithm tries to add a mutual edge. If such an addition would violate the R-regularity property, it triggers a local pruning process on node js neighborhood to restore degree bounds while preserving the quality of the graph.

2.1 Graph Initialization Analysis

The initialization phase of the Filtered Vamana algorithm traditionally begins with an empty graph structure. To evaluate potential performance optimizations, we investigated the impact of initializing the graph with random edges instead of an empty structure. Our analysis employed the same performance metrics as the previous investigation: **Memory Usage**, **Recall Percentage**, **Query Speed**, and **Graph Construction Time**. The experimental methodology involved utilizing the 'enforceRegularity' function to ensure each vertex maintained a minimum degree of R, consistent with the constraints employed in the standard Vamana Indexing Algorithm.

The empirical results revealed several significant findings. Most notably, the R-Regular initialization approach demonstrated a substantial increase in memory consumption, attributable to the expanded adjacency lists necessitated by the additional initial edges, as documented in Table 1. Despite the implementation of R-Regular initialization, and even with the additional optimization of edge filtering based on node similarity, the improvement in recall percentage remained marginal, not exceeding 0.5%. Query performance remained statistically unchanged, while Graph Construction Time showed a minor degradation, as illustrated in Figure 3.

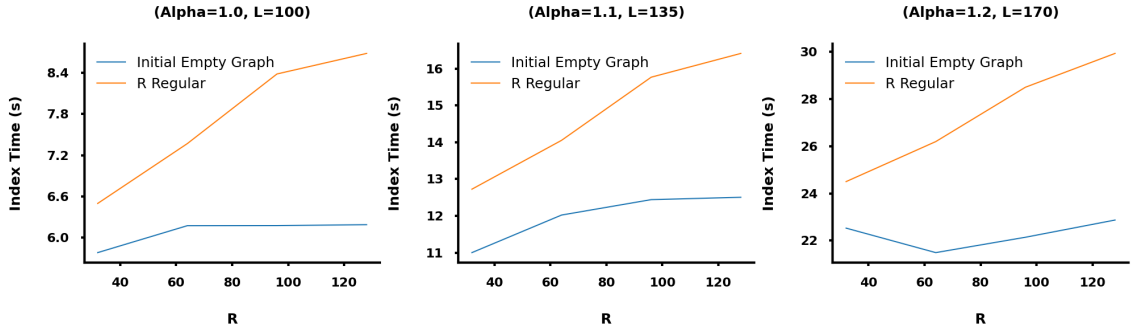

Figure 3. Analysis of Index Construction Time as a function of R across various hyperparameter configurations. The results demonstrate consistent performance degradation with R-Regular initialization due to the overhead of initial edge insertion, with the Empty Graph initialization maintaining superior performance across all tested R values.

Memory of Empty Graph	Memory of R-Regular Graph	R value
22MB	24MB	32
27MB	40MB	64
29MB	50MB	96
30MB	70MB	128

Table 1. Comparative analysis of memory utilization between empty graph initialization and R-Regular graph initialization approaches

Based on these experimental findings, we concluded that the empty graph initialization approach remains optimal, and subsequent experiments maintained this initialization strategy rather than pursuing the R-Regular approach.

2.2 Parallelization Implementation and Analysis

To enhance the computational efficiency of the Filtered Vamana algorithm, we investigated parallel processing strategies to exploit modern multi-threading capabilities. The initial algorithm structure presented inherent challenges to parallelization due to edge manipulation dependencies within the graph. However, analysis of our single-filter-per-node implementation revealed that the Filtered Vamana algorithm naturally decomposes into independent subgraphs corresponding to distinct filters, presenting an opportunity for parallel execution.

We developed a parallelization strategy that restructures the node iteration process: rather than random traversal across all graph nodes, we implemented a two-level iteration scheme. The primary iteration occurs across distinct filters, while secondary processing assigns dedicated threads to handle points associated with each filter. This approach ensures thread safety as each thread operates exclusively on its assigned filter's subgraph, eliminating potential race conditions in edge modifications. The implementation utilized the OpenMP framework, selected for its efficient handling of parallel loop constructs and straightforward integration with our existing architecture.

Experimental evaluation of the parallel implementation demonstrated several significant findings. The algorithmic correctness remained preserved, as evidenced by unchanged recall percentages across all configurations. Query performance, which remained sequential in the current implementation, maintained consistent performance metrics, presenting a potential avenue for future optimization. Memory utilization exhibited the expected linear

scaling with respect to thread count, while substantial performance improvements were observed across all hyperparameter configurations, as illustrated in Figure 4. Notably, the parallelization scheme demonstrated favorable scaling characteristics with increasing dataset sizes, indicating robust performance for larger-scale applications **??**.

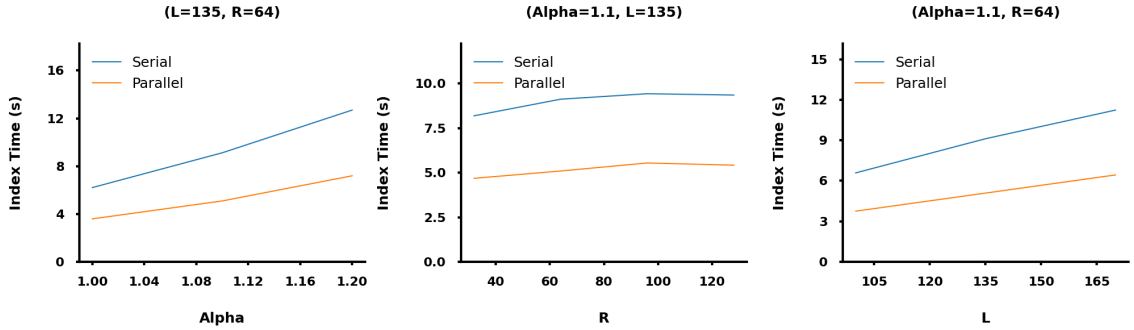

Figure 4. Comparative analysis of Index Construction Time between serial and parallel implementations of the Filtered Vamana algorithm across varied parameter settings. The performance improvement remains consistent across the spectrum of L and R values, demonstrating robust scalability of the parallel implementation.

Size of Dataset	Execution Time of Serial	Execution Time of Parallel	Speedup Percentage
10k	8.91741s	5.02602s	77%
1M	2750.09s	2094.58s	31%

Table 2. Comparison of Serial and Parallel implementation for different Dataset Sizes with $alpha = 1.1$, $R = 64$, $L = 135$

3 Stitched Vamana

The Stitched Vamana algorithm offers an intuitive approach to implementing Vamana in a filtered setting, as proposed in the same research paper introducing Filtered Vamana. This algorithm constructs the graph by iterating over all possible filter values, creating a subgraph for each filter, and applying the standard Vamana algorithm to construct an indexing graph specific to that filter. Once each subgraph is built, its edges are integrated into the original graph, effectively *"stitching"* them together. After all subgraphs are combined, a `robustPrune` operation is performed across all vertices in the graph to maintain regularity and eliminate redundant edges.

In our implementation, we initialize an `ANN` object for each filter iteration. For each filter, we insert all points associated with that filter into this smaller `ANN` object, apply the Vamana algorithm to construct the subgraph, and finally, merge the resulting edges back into the main `ANN` object.

3.1 Graph Initialization Analysis

Following the methodology established in the Filtered Vamana analysis, we conducted a comparative study of R-Regular versus Empty graph initialization approaches, evaluating their impact on four key performance metrics: **Memory Usage**, **Recall Percentage**, **Query Speed**, and **Graph Construction Time**.

As anticipated, the experimental results demonstrated a consistent increase in memory utilization correlating with higher values of the R hyperparameter, as illustrated in Figure

5. This increased memory footprint is attributed to the expanded adjacency list requirements of the graph structure. Notably, unlike the Filtered Vamana implementation, the addition of initial edges yielded significant improvements in recall performance (Table 3). This enhanced performance characteristic is particularly pronounced in the Stitched Vamana algorithm, potentially attributable to the architectural independence of individual subgraphs within the system.

Furthermore, the analysis revealed a consistent reduction in index construction time, ranging from 0.8 to 1.2 seconds across all parameter configurations. This improvement can be attributed to the reduced computational overhead within the Vamana function during the R-Regular construction of individual subgraphs. The empirical evidence suggests that the combination of improved recall performance and reduced index construction time makes the R-Regular initialization approach a viable optimization strategy for the Stitched Vamana algorithm. Consequently, subsequent experimental investigations will incorporate this optimization technique.

Recall of Empty Graph	**Recall of R-Regular Graph**	R **value**
90.1	95.6	32
96.8	98.0	64
97.6	98.3	96
97.7	98.5	128

Table 3. Comparative analysis of recall performance (unfiltered queries) between empty and R-Regular graph initialization approaches in the Stitched Vamana algorithm.

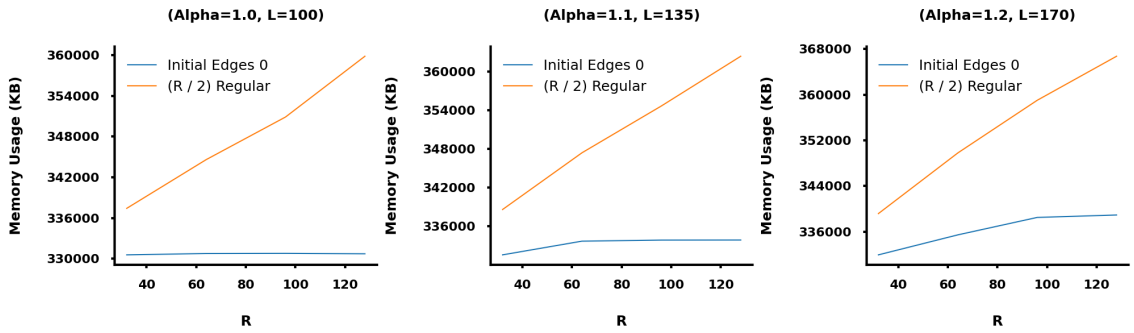

Figure 5. Analysis of memory utilization as a function of R across various parameter configurations. The results demonstrate a near-linear increase in memory consumption with R-Regular initialization, consistently exceeding the memory requirements of empty graph initialization across all tested values of R.

3.2 Parallelization Implementation and Analysis

The Stitched Vamana Algorithm's inherent structure, with its filter-based partitioning and independent subgraph construction, presents natural opportunities for parallel execution optimization. We enhanced the algorithm's parallel processing capabilities by restructuring the robustPrune vertex iteration methodology. Our implementation adopts a two-tier iteration strategy: a primary iteration across distinct filters, followed by a secondary iteration over filter-specific vertices. This approach leverages OpenMP's parallel processing capabilities while maintaining the algorithm's fundamental characteristics. The critical aspect of this implementation is the preservation of subgraph independence, ensuring that concurrent thread operations remain isolated and free from inter-thread interference.

Empirical evaluation of the parallel implementation demonstrated preservation of algorithmic integrity, with recall percentages remaining stable within statistical margins (±0.1%) across multiple experimental runs. Memory utilization exhibited the expected linear scaling with respect to thread count, consistent with theoretical predictions. As illustrated in Figure 6, the implementation achieved significant reductions in Index Construction Time across all parameter configurations.

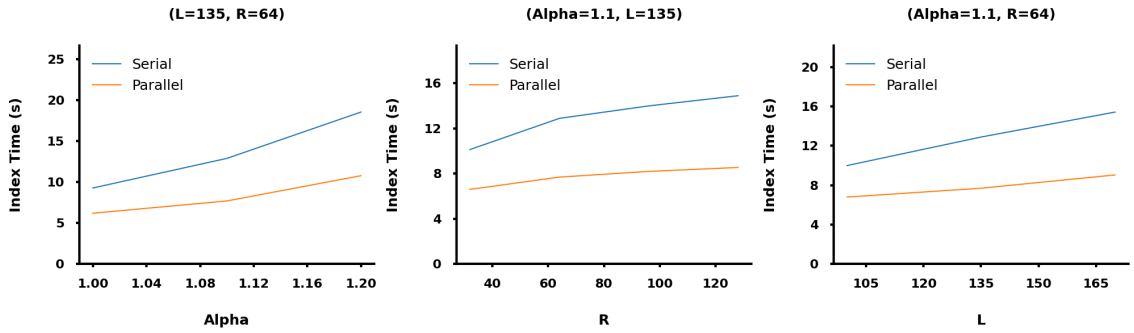

Figure 6. Performance analysis of Index Construction Time comparing serial and parallel implementations of the Stitched Vamana algorithm across various parameter configurations. Results demonstrate consistent performance improvements and effective scaling characteristics with increased dataset dimensions.

4 Discussion

Based on the report we conducted we have come to the following results

(1) Vamana Index Algorithm: Robust Performance measured through the Total Recall Percentage independent of hyper parameter settings which makes it a viable option when an exhaustive search in the hyperparameter space is not feasible. However we can see that the time complexity of the algorithm and the absence of possible parallelization sections makes scaling to larger datasets harder and requires increased computational power.

(2) Filtered Vamana Index: The ability of the algorithm to handle filtered queries and provide k-Nearest Neighbour results with added requirements does not hinder it when handling unfiltered queries. In our experiments due to limited resources in terms of memory and time, Filtered Vamana was the only algorithm that was able to finish execution on datasets with larger sizes (>1 Million) showcasing its efficiency in terms of memory and time usage. However for unfiltered queries we noticed a slight dependance on the hyperparameters and it's recall for those queries was sensitive to changes in the settings.

(3) Stitched Vamana Index: Although implementation of this variation was significantly easier, the usage of independent subgraphs proved to have less accuracy than the Filtered Vamana approach, however we had increased query speed showcasing that the graphs constructed did not have as many redundant edges. Because of the way we implemented the subgraphs it was not feasible have parallel execution for larger datasets due to memory constraints. Further experimentation could be conducted with optimized subgraph construction and stitching.