Exploring Optimizations on a Generic Implementation of Vamana-Based Indexing Algorithms

Γεωργίου Ανδρέας
1115 2021 00218

Θεοφυλάκτου Αλέξανδρος
1115 2021 00220

Θεοφυλάκτου Κωνσταντίνος
1115 2021 00221

January 13, 2025

Contents

Abstract

This paper reviews the results of specific optimizations and modifications on a specific implementation of the graph-based Vamana Indexing Algorithm [9] for finding the K-Nearest Neighbors in both Filtered and Unfiltered Datasets. Four different optimization goals are considered during the optimization process. Index Creation Time Minimization, Query Time Minimization, Recall Score Maximization, and Graph Density minimization. We opt to provide a simple, generic interface for interacting with a set of three Vamana Based Algorithms, that work well with any datatype under minimal requirements. However, specific optimizations are made for particular datasets of certain and common datatypes. Experiments were done on a selection of various optimization configurations, each focused towards one or more of the four objectives. The results aim to highlight the trade-off between isolated optimizations, and emphasize the importance of application-specific selection of optimizations.

1 Introduction

Despite their simplicity, K-Nearest Neighbor (K-NN) algorithms play a vital role in various domains, including data analysis, recommendation systems, and pattern recognition. However, the naïve implementation of K-NN algorithms becomes increasingly inefficient as dataset sizes grow, making it unsuitable for large-scale applications. To address this challenge, the original authors introduced the Vamana Indexing Algorithm, which enables fast and accurate billion-point nearest neighbor searches on a single node, offering significant improvements in scalability and efficiency. [9]

While these foundational works of the authors present a robust mathematical and algorithmic framework, they leave significant flexibility in implementation details, such as the choice of data structures to replace mathematical sets. The selection of appropriate data structures is crucial, as it can dramatically improve algorithmic efficiency. For example —one that we have also incorporated in our implementation—utilizing a heap structure instead of a set-like structure for maintaining a dynamically sorted list during insertions can yield significant performance gains.

Beyond thoughtful data structure selection, our work explores additional optimizations to enhance the performance of Vamana-based indexing. Common algorithmic optimization techniques, including parallelization, approximation methods, caching, memoization, and SIMD instructions, some of which were evaluated in this work. Moreover, we modified the original algorithms to address datasets where data points belong to at most one category while excluding range-filtered queries. These modifications enable straightforward per-category parallelization, which in most cases significantly accelerates the creation of filtered indices, and speeds up the querying process.

To evaluate our implementations, benchmark tests were conducted on various combinations of selected optimizations across three proposed index creation algorithms. The benchmarks include an index creation phase using a dummy dataset of 10 000 points, followed by an index querying phase with 10 000 queries. Accuracy was measured using recall scores, calculated against exhaustively computed ground truth, to ensure rigorous performance evaluation.

Using the ablation methodology [7], we assessed the contribution of individual optimizations, highlighting trade-offs between accuracy and performance. The optimal optimization sets for each index type were identified and further tested on a larger dataset of 1 000 000 points, to analyze their scalability and effectiveness under significant computational load.

The remainder of this paper is organized as follows: Section 2 describes the general structure and provides usage instructions. Sections 3 and 4 briefly outline the modifications and specific implementations of each index type. Section 5 explores the applied optimizations, while Section 6 explains the benchmark methodology and presents detailed results for each optimization set. Finally, Section 7 provides conclusions and future directions.

2 General Structure

This section provides an overview of the decided structure of the code, along with instructions for correct usage. For the development of our implementation we have exclusively used C++17, along with a range of its standard libraries, focusing primarily on the Standard Template Library (STL). The main object of focus is the DirectedGraph template class, which includes a broad variety of methods, that make use of generic template functions for their implementation.

The main program is modular and can execute different sections of code depending on the arguments given through the command line. The use of the main program is to read data from a file and build/load a Vamana Index which can also be stored. Then, the index can be queried to retrieve the approximate K-Nearest Neighbors of a query point with very high accuracy.

The `interface.hpp` header file wraps the project's dependencies, and acts as an interface library, providing the user with functions that are designed to be interacted with. The `main.cpp` file includes the interface header and is therefore granted access to the entirety of the other header files.

2.1 Structures Explained

2.1.1 Id

The Id class is simply an extension of the native `int` data type. The purpose of the extension is to initialize any uninitialized Ids with -1, which is considered the invalid Id. This makes error checking and prevention possible in cases where an Id was not initialized correctly.
The Id structure is part of the Node and Query structures, and is treated as an integer. Its purpose in the Node and Query structures is to provide a more lightweight copy-able index to the actual Nodes or Queries, while also being easy to hash in order to leverage the hashable type benefits such as instant access.

2.1.2 Node

The Node class is the internal structure used in the DirectedGraph class, which is the main object of focus in this implementation. The Node structure represents a vertex in the graph. It provides each datapoint with a unique Id, while also separating the category/filter information from the actual data.
The Node structure is hashable with an extension of the `std` namespace. The Node's hash function utilizes the `std::hash<int>` native hash function, which is simply used on the `Node.Id` field. By doing so, we enable the Node to be part of unordered containers that use hash functions, and we therefore take advantage of the benefits they provide.

2.1.3 Query

Similar to the Node Class, the Query class is used as an encapsulation of the query data, with additional fields being the Id, the Category, and a Filtered Flag.
The Query structure is also hashable with an extension of the `std` namespace, and is implemented by hashing the Id, just as the Node.

2.1.4 DirectedGraph

The DirectedGraph template class is the main object of focus. It implements a Directed Graph Data Structure. Datapoints are encapsulated in the Node structure which represents a vertex of the graph. All the vertices are stored in an `std::unordered_set` structure that uses the `std::hash<Node>` function. In doing so, we can ensure $O(1)$ access. Deletions are not supported because they are not required, and insertions only happen once, before the Vamana Indexing Algorithm is performed.

The outgoing edges of the graph are stored in an `std::unordered_map` with key the Id and value an unordered set of Ids. To avoid using excess space, any vertex with no outgoing edges does not have its Id as a valid key to the edges map. We use the function `mapKeyExists` to avoid accessing a non-existent value with an invalid key. On the occasion where the last neighbor of a vertex is removed, and therefore the set of neighbors becomes empty, the key is also deleted from the map. Once again, the `std::hash<int>` native hash function is used on the Id for $O(1)$ retrieval of the Ids of the neighboring nodes, which can then be accessed in $O(1)$ from the Nodes unordered set.

To add a point to the graph, the `createNode` method must be used. It is the user's responsibility to split the category from the actual data in the case that they are all joined in a single vector. To add or remove an edge between two nodes, the `addEdge` function is used, which takes as arguments the Ids of the two nodes in a from-to order.

2.2 Supported Data Types and Requirements

Our implementation is dynamic and works with any datatype that meets the following requirements:

- Overloaded `less <` operator
- Overloaded `equality =` operator
- Overloaded `ostream <<` operator (optional, required if store/load are to be used)
- Overloaded `istream >>` operator (optional, required if store/load are to be used)
- Distance Function for specific datatype (complete your own in `util.hpp`).
- isEmpty Function for specific datatype (complete your own in `util.hpp`).

By setting the value option `-distance 3` the program is overloaded to use the desired custom type. It is essential that the complete datatype is visible in the `util.hpp` header.

The datatype is then encapsulated in the Node and Query structures. The Node and Query structures are overloaded as needed, and in their internal implementations make use of the custom datatype's overloads.

For the simple datatype of `std::vector<T>`, there exist already implemented distance and isEmpty functions. It is important to note that to use our distance functions, the `T` type encapsulated in the vector should have all the appropriate overloads used in our distance functions (implemented in `util.hpp`).

There are some interface requirements in order for the interface functions to be used correctly. There must exist a function that reads data and queries from a file depending on the specific format. There already exist such functions for specific formats such as `.vecs` and `.bin` as described in the SIGMOD Contest 2024 [1].

2.3 Instructions for use

The user primarily interacts with the Command Line Interface (CLI), the `interface.hpp` header, and if required, a configuration and a utilities file if they desire to make the implementation specific for their own complete data type. Instructions for type specification lie in section 2.2 and inside the code itself.

2.3.1 Building and Running the Project

In order to run the main program to build the Index and retrieve the K-Nearest Neighbors, it must first be compiled. For the compilation, any C++ compiler that supports C++17 will do.

For the Compilation a simple Makefile is used that simplifies the compilation process. From the project's root directory, and from a shell execute the `$make` command. This will build the main object file, and will also produce the main binary in the `bin/` folder. The main executable is named `main`, and its full relative path to the root directory is `./bin/main`.

To execute the program, simply call `$./bin/main <argument list>`. The argument list will be further explained in section 2.3.2.

The Makefile can also be used to run some simple pre-determined scenarios, to avoid providing a lengthy argument list on each execution and also prevent any mistakes during the process. Some Makefile shortcuts include:

- `$make run_v` runs Scenario 1 for the Vamana Index. Scenario 1 has all the default values of the vamana index, and gets its data from the following files:
  - data: `"data/siftsmall/siftsmall_base.fvecs"`
  - queries: `"data/siftsmall/siftsmall_query.fvecs"`
  - groundtruth: `"data/siftsmall/siftsmall_groundtruth.ivecs"`
- `$make run_f` runs Scenario 2 for the Filtered Vamana Index. Scenario 2 has all the default values of the filtered vamana index, and gets its data from the following files:
  - data: `"data/contest-data-release-1m.bin"`
  - queries: `"data/contest-queries-release-1m.bin"`
  - groundtruth: `"data/contest-groundtruth-custom-1m.txt"`
- `$make run_s` runs Scenario 3 for the Stitched Vamana Index. Scenario 3 has all the default values of the stitched vamana index, and gets its data from the following files:

- data: `"data/contest-data-release-1m.bin"`
- queries: `"data/contest-queries-release-1m.bin"`
- groundtruth: `"data/contest-groundtruth-custom-1m.txt"`

2.3.2 Command Line Arguments

Our program provides the option to dynamically alter its functionality through command line arguments.

The command line arguments are split in two categories:

1. **Value Options:** A Value option consists of two parts. The option, and its value. Options are preceded by a single dash `-`, and refer to a specific global variable. Options are followed by the appropriate value.

   Example of a Value Option: `-nthreads 8`

2. **Flags:** A flag can either be set or not set. Specifying the option in the CLI will set the flag. Flags are preceded by double dashes `--`.

   Example of a Flag: `--debug`

The CLI arguments are parsed and the appropriate values are stored in a static global structure visible to all parts of the program. Almost all arguments are optional, except from the index type. Any non-specified arguments will default to specific values based on the index type. All the options will be explained in the following exhaustive list.

Required Option (choose only one of the following):

- `--vamana`: Sets the index type to Vamana.
- `--filtered`: Sets the index type to Filtered Vamana.
- `--stitched`: Sets the index type to Stitched Vamana.

Optional Arguments and their Default Values: (Any other argument is not acceptable and will throw an invalid argument error)

Value Option	Description	Default Vamana	Default Filtered	Default Stitched
`-k`	Number of neighbors	100	100	100
`-L`	Maximum cardinality of the Candidate Set	100	100	100
`-R`	Maximum out-degree	14	14	64
`-a`	greedy coefficient a	1	1	1
`-t`	threshold - percentage in (0,1]	0.5	0.5	0.5
`-Rs`	Rsmall (for stitched Vamana Index)	-	-	32
`-n_data`	Number of data points	10000	1000000	1000000
`-n_queries`	Number of query points	100	10000	10000
`-n_groundtruths`	Number of groundtruth points	100	10000	10000
`-dim_data`	Dimension of the data vectors	-	102	102
`-dim_query`	Dimension of the query vectors	-	104	104
`-store`	Location to store the index	""	""	""
`-load`	Location of stored index	""	""	""
`-data`	Location of the Data File	Scenario 1	Scenario 2	Scenario3
`-queries`	Location of the Queries File	Scenario 1	Scenario 2	Scenario 3
`-groundtruth`	Location of the Groundtruth File	Scenario 1	Scenario 2	Scenario 3
`-n_threads`	Number of Threads	1	1	1
`-distance`	Distance Function (See supported)	1	1	1
`-extra_edges`	Add Random Extra Edges after Index Creation	0	0	0
Flag	**Description**	**Default Vamana**	**Default Filtered**	**Default Stitched**
`--debug`	Displays log messages on the console			
`--stat`	Gathers performance data and stores in file = argv			
`--only_unfiltered`	Discards any filtered Queries			
`--only_filtered`	Discards any unfiltered Queries			
`--data_unfiltered`	Treats data as if unfiltered (discards categories)	✓		
`--no_create`	Skips index creation phase (auto set if `-load`)			
`--no_query`	Skips querying phase			
`--random_start`	Random point as start in Greedy Search			
`--pqueue`	Priority Queue instead of Unordered Set for $\mathcal{L}$			
`--no_rgraph`	Skips Rgraph initialization (only for Vamana)			
`--acc_unfiltered`	Queries each category individually and accumulates results			

Table 1: CLI List

3 Vamana Graph Index

3.1 Greedy Search

Our implementation of the Greedy Search Algorithm [9] returns a pair of two `std::unordered_set`, containing the node Ids of the K closest points from $\mathcal{L}$ and visited nodes respectively. The algorithm follows the original paper [9], with a few optimizations on the data structures used. These optimizations are further discussed in detail in section 5.2. More optimizations such as thread parallelization—discussed in section 5.4—require for GreedySearch to be made thread-safe. More details in section 5.4.2.

3.2 Robust Prune

The Robust Prune implementation follows the original paper [9] to effectively contain a node's out-degree to at most R. For the set union action, we directly insert the elements into the set. The `DirectedGraph::clearNeighbors` method removes all the out-going edges for a specific node. When iterating over the nodes in the candidate set V, we use STL's `std::iterator` structures that enable for safe set traversal with concurrent deletions.

Upon further optimizations, robustPrune was required to be made thread-safe. For this reason—apart from the necessary synchronization structures—the implementation was further modified. These modifications are further analyzed in section 5.4.2.

3.3 Vamana Algorithm

The Vamana Algorithm initializes the graph with random edges—a step that might as well be omitted—then proceeds to finalize the index by following the Vamana Algorithm [9], combining the graph's methods alongside some utility functions. The starting node for the greedy search inside the vamana algorithm is decided based on the command-line arguments, and can either be the medoid of the dataset or a random data point. Further discussion about the Starting Point Selection can be found in section 5.5.1. A minor modification to the original algorithm was inserting the neighbor σ_i into the neighbors of j in every occasion, and letting Robust Prune remove it if it wasn't needed. That is because σ_i is passed into the candidate set alongside the neighbors of j, which is the same as directly passing the neighbors of j as the candidate set, and thus skipping an unnecessary check.

Upon tackling the challenging task of parallelizing the Vamana Algorithm, further modifications were made. More details in section 5.4.2.

4 Vamana Graph Index with Categories and Filters

The Filtered DiskANN [5] extends the works of the original DiskANN [9] who first introduced the Vamana Indexing Algorithm. Two different algorithms were proposed to tackle the task of finding the approximate K-Nearest Neighbors under specific filters. One algorithm is called the Filtered Vamana Algorithm, and the other is called the Stitched Vamana Algorithm.

The Filtered Vamana Indexing mimics the original Vamana Indexing Algorithm, with adaptations for filters to all of its components. For this reason, there are proposed adaptations of both the Greedy Search and the Robust Prune to handle filters and categories, named Filtered Greedy Search and Filtered Robust Prune respectively.

The Stitched Vamana Indexing algorithm makes heavy use of the original Vamana for unfiltered data. It works by creating separate subgraphs for each category present in the dataset, and populating the graph with any points belonging to that category. Then, the subgraphs are stitched together and the total graph is pruned.

In our implementation, we follow the lines of these proposals under some strict **assumptions**:

- No point in the dataset can belong to more than one category
- All queries performed in the index are either unfiltered, or simple categorical.

During the creation of nodes in the graph, we keep track of all the categories we encounter, and create a map with the Ids of the nodes belonging in that category. This step simplifies computations in later parts, where per-category distinction is required, such as the starting point selection for a Filtered Greedy Search.

Once again, the authors [5] suggest careful selection of the starting point for the searches, and they even provide a method for reduced-stress medoid approximation. However, their method heavily relies on points belonging in multiple categories. Their proposed method involves a threshold parameter, which we got inspired from and slightly altered to better fit our own implementation.

The threshold parameter in our implementation controls the percentage of nodes to be sampled from each category, before identifying the medoid. This percentage parameter plays a vital role in a balanced medoid approximation for categories of various sizes. In contrast, the original implementation assumed a fixed threshold t that did not account for the size of each category.

4.1 Filtered Greedy Search

The filtered greedy search largely follows the implementation described in the original paper [5], but with significant modifications due to the absence of set operations required for overlapping categories. The algorithm starts with a single designated starting node rather than a set of starting nodes. The single node can either be the medoid of the query's category, or a randomly selected point from that same category.

Identically to the Greedy Search, two implementations are available: one using unordered sets and another employing a priority queue. The choice of implementation depends on the user's preference, with further optimizations and performance trade-offs discussed in Section 5.2.

In contrast to the Greedy Search [9], the Filtered Greedy Search is not required to be thread-safe, and therefore no further modifications were made. The explanation for this can be found in Section 5.4.2, under the parallelization of the Filtered Vamana Algorithm.

4.2 Filtered Robust Prune

The filtered robust prune follows the algorithm described in the original paper [5], but modified under the assumption that each node belongs to at most one category. This restriction eliminates the need for complex set operations required to handle overlapping categories.

Similar to the Filtered Greedy Search, the Filtered Robust Prune is also not required to be thread-safe, and therefore also avoids employing locking mechanisms for synchronization.

4.3 Filtered Vamana Algorithm

The Filtered Vamana Algorithm stays close to the original implementation [5], but was slightly altered to adhere to the Filtered Greedy Search modifications regarding the starting node. Identically, the minor modification regarding σ_i—described in section 3.3—was also done in this case.

The parallelization of the Filtered Vamana Algorithm is quite straightforward, as can be seen in section 5.4.2.

4.4 Stitched Vamana Algorithm

The Stitched Vamana Algorithm makes great use of the Vamana Algorithm's implementation. It begins by populating an independent graph with nodes from a specific category, associating them with their original indices for later integration into the main graph. To optimize memory usage, the same subgraph variable is reused across all categories, ensuring no additional memory overhead for the management of multiple subgraphs. The Vamana Algorithm [9] is executed on the populated subgraph.

After constructing the subgraph for a category, its edges are stitched back into the main graph. The stitching process involves transferring all edges from the subgraph to the main graph while ensuring that each edge is accurately mapped to its original node Id. Additionally, the medoid of the category is computed during the Vamana Algorithm, and we store it for further use to avoid recalculation. Once the stitching is complete, the subgraph is cleared and reused for the next category.

The implementation also supports a parallel version of the Stitched Vamana Algorithm, allowing for per-category parallelization. This is further discussed in section 5.4.2.

5 Optimizations

5.1 Compiler Optimizations

To enhance the performance of our implementation, we leveraged compiler-level optimizations by enabling the `-O3` flag during compilation. The `-O3` flag applies aggressive optimizations which can result in increased compilation time, larger binaries, and floating point precision errors. However, these trade-offs align with the goals of this project as follows:

- Compilation Time and Binary Size: Not a priority for this project.
- Floating Point Inaccuracies: The minor precision errors introduced by the `-O3` flag are acceptable in the context of the Vamana Index, an approximate method for finding the K-Nearest Neighbors (Approximate K-NN). The method does not require exact precision, especially when the recall score remains near the results that where achieved in the original paper [9].

By leveraging the `-O3` compiler optimization we further increased computational efficiency of our implementation. The benefits of compiler optimizations are general-purpose, widely-understood, and not specific to the Vamana Indexing Algorithm. For this reason, we have

chosen to omit presenting the detailed performance comparison, to maintain focus on more important, task-specific optimizations.

5.2 Priority Queue for $\mathcal{L}$ in Greedy Search

The authors of the Disk-ANN [9] provide sufficient flexibility in the choice of data structure to replace the mathematical sets used in their algorithmic formulations. The naïve approach would involve implementing the algorithms using sets and performing set operations as outlined. However, upon closer examination of the Greedy Search Algorithm [9], it becomes evident that using a **Priority Queue** for the $\mathcal{L}$ set could offer significant benefits.

The candidate set $\mathcal{L}$ is continuously updated to retain the (L) nearest neighbors of the query point. In the naïve set-based implementation this can be achieved either by iterating through the entire set tracking the L closest points, or by sorting the elements of the set by the closest distance from the query point. Our initial implementation employed the `DirectedGraph::_closestN` method which relies on the `std::nth_element` function [4] along with a lambda custom comparator. The `std::nth_element` function promises $O(|\mathcal{L}|)$ time, where $|\mathcal{L}|$ is the size of the candidate set before cropping.

Instead of repeatedly sorting and cropping $\mathcal{L}$ on every iteration, we implement the candidate set using an `std::priority_queue` as a **maxHeap** with a **maximum size of** L. Although a maxHeap may seem counter-intuitive for finding the minimum distances, it significantly reduces the number of operations required to maintain $\mathcal{L}$ within its size limit.

The maxHeap allows for efficient operations:

- **Insertion:** When $|\mathcal{L}| < L$ is performed in $\Theta(\log(|\mathcal{L}|))$ time.
- **Conditional Insertion/Replacement:** Once $|\mathcal{L}| = L$, the farthest point (top element of the heap) can be compared against a new candidate:
  - If the candidate's distance is greater than the top element's distance, it is skipped.
  - If the candidate's distance is smaller, the top element is popped in $\Theta(1)$ time, and replaced with the candidate in $\Theta(\log(L-1))$ time.

Performance Analysis

After carefully gathering data on the candidate set sizes on each timestep, and following the documented promised complexities, we compare the cost per timestep for each data structure. The results are summarized in Figure 1.
The results show that the **Priority Queue consistently outperforms the Unordered Set** by a large margin in terms of the cost per time step, except from the final step. The spike in the cost at the final step arises because extracting the top K elements from the Priority Queue requires popping each element individually. However, since this step is performed only once, its impact on the total computational cost is negligible.

Additionally, the Priority Queue implementation **accelerates the convergence of the Greedy Search** Algorithm compared to the Unordered Set. This improvement can be attributed to the sorted nature of the Priority Queue, which maintains an ordered structure throughout the search. In contrast, the `closestN` approach returns the elements in an unordered manner.

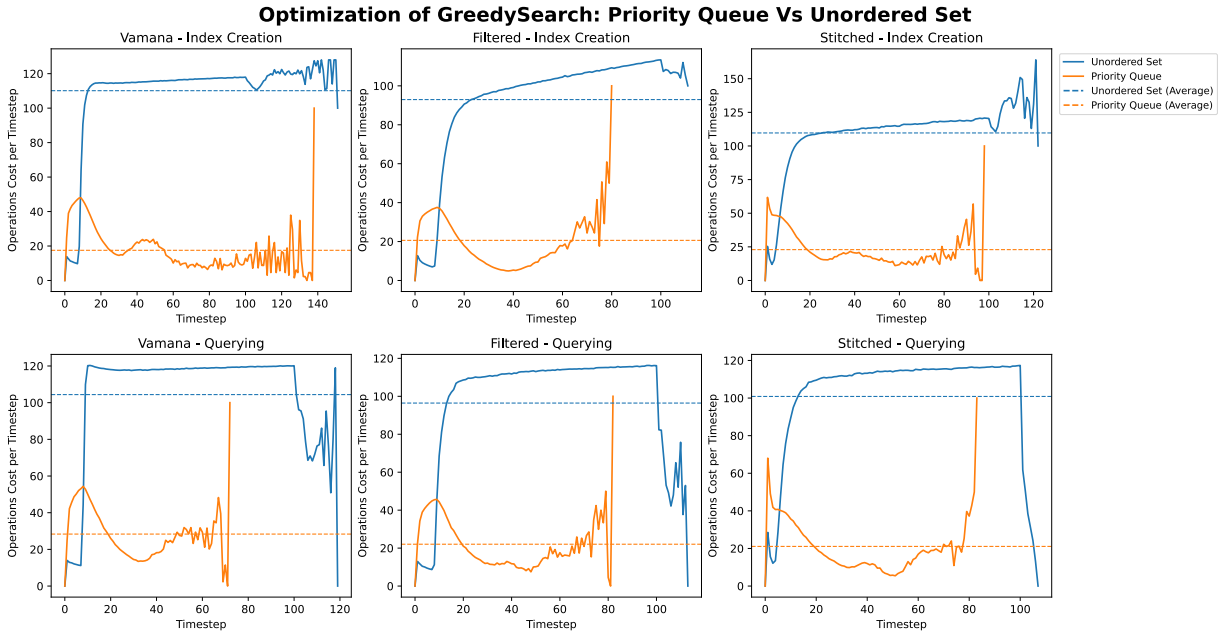

Figure 1: Priority Queue Vs Unordered Set implementation for $\mathcal{L}$ in Greedy Search

Beyond reducing computational overhead, the results of the benchmark tests—as detailed in Section 6—indicate that the use of a **Priority Queue improves the recall score**. This improvement suggests that maintaining an ordered candidate set positively impacts the quality of the search results.

5.3 Distance Function optimizations

The performance of any K-Nearest Neighbor (K-NN) algorithm is critically dependent on the efficiency of the distance function, which is the most fundamental, frequently-called operation when facing such a task.

Although the implementation described in this work is designed to support generalized data types, its efficiency must be evaluated under specific scenarios. As detailed in Section 6, the datasets used for this study consist of vectors represented by floating-point numbers. For such data, the **Euclidean Distance** is the most appropriate distance function [12].

5.3.1 Naïve Euclidean Distance without Square Root:

Our first approach in optimizing the Euclidean Distance involves calculating the sum of squared differences, but omitting the calculation of the square root since the distances are only used for comparisons. Since the square root function is strictly increasing ($\sqrt{x} < \sqrt{y}, \quad \forall x, y \ : \ x < y$) it does not alter the relative order and can therefore be omitted for a computational benefit without any consequences on the recall score.

This optimization was implemented very early during the development process, and since its benefit is well documented it will not be evaluated in contrast to truly naïve implementations that calculate the square root. Instead, this will serve as the default, for other distance functions to compare against.

5.3.2 SIMD Euclidean Distance:

This optimization significantly reduces the time required for Euclidean distance calculations by leveraging SIMD (Single Instruction Multiple Data) operations implemented through compiler intrinsics and C++ native libraries. The approach utilizes assembly-level optimization with 256-bit pseudo-registers, allowing eight floating-point values to be loaded simultaneously into a single register [8].

The subtraction and multiplication operations are performed between two registers (one for each data point). The results are accumulated in another register. Finally, the accumulated results are stored in an array of 32-bit floating-point numbers, and the elements of this array are summed to compute the total distance.

While this optimization drastically reduces the overall computational load, it does not come without trade-offs. The implementation is highly specific to vectors of either 128 or 100 floating-point numbers, limiting its flexibility for other vector sizes, or other data types.

Breakdown of the optimization:

1. Vectors of 128 dimensions:
    - 16 operations using 256-byte registers
    - 8 operations for summing intermediate results
    - **Total:** 24 operations
2. Vectors of 100 dimensions:
    - 12 operations using 256-byte registers
    - 1 operation using 128-byte register for the remaining 4 dimensions
    - 8 operations for summing intermediate results of the 256-byte register operations
    - 4 operations for summing intermediate results of the 128-byte register operations
    - **Total:** 25 operations

5.3.3 Threaded Parallel Euclidean Distance:

This attempted optimization involved distributing the calculation of the Euclidean Distance among many threads. However, the overhead of launching and joining threads proved to be too significant in contrast to the relatively low workload, especially for a function that is called so frequently in this project. As a result, calculations with this distance function led to extremely long calculation times.

This optimization was immediately discarded and is not included in our experimental evaluations, due to its results not being nearly competitive enough, even in comparison to the naïve approach.

5.4 Thread Paralellization

Efficient utilization of computational resources is essential for effectively tackling large scale tasks such as finding the approximate K-Nearest Neighbors. Thread parallelization is an optimization technique that leverages the multi-threading capabilities of multi-core CPUs to distribute the computational load among multiple cores, significantly reducing execution time. However, effective multi-threaded implementations can not exist for any task [10].

5.4.1 Limitations

The task of parallelizing an algorithm designed for a single thread can be quite challenging. Proper thread synchronization and effective load balancing are crucial factors in the effectiveness of this optimization. Synchronization structures—like mutexes and condition variables—are used to ensure proper thread synchronization, and simple scheduling schemes are applied to avoid thread starvation.

As seen in section 5.3.3, launching threads for a simple task can actually decrease the overall efficiency of the program, due to the overhead required to initialize and maintain the appropriate structures.

5.4.2 Specific Implementations

In the context of our project, multi-threaded implementations were implemented in various stages—some more straightforward than others. Synchronization structures such as Locking mechanisms and condition variables were required for some more complex and challenging parallelization of certain algorithms—the most challenging one being that of of the Vamana Indexing Algorithm.

Parallel Medoid

One of the most straightforward implementations is the parallel medoid because it does not use any locks, relying instead on read-only access to shared data, and the workload is evenly distributed among threads.

The workload in our medoid implementation is divided by splitting the dataset of nodes into chunks, with each thread processing a specific range of indices. The chunk size is calculated based on the number of threads, and any remaining nodes are evenly distributed among threads to balance the workload. Each thread independently computes the total distance for its assigned nodes and keeps track of its own local minimum distance and corresponding node. Synchronization is achieved by ensuring that threads access shared datasets in a read-only manner and write their results to thread-local storage. This eliminates the need for explicit synchronization mechanisms. Once all threads complete their computations and join, the results are aggregated to identify the global medoid.

Parallel Rgraph

The parallel implementation of the `DirectedGraph::Rgraph` method divides the workload among multiple threads, with each thread responsible for processing a subset of nodes. Each thread takes an index by the shared index variable, increments the shared index for the next thread to take, and then unlocks the mutex to allow other threads access. A mutex is used to ensure thread-safe access to this index and to synchronize the addition of edges, preventing race conditions when modifying the `DirectedGraph::Nout` map.

Parallel Vamana

The parallel Vamana algorithm uses the same shared index approach as the parallel Rgraph. To safely access and modify the `DirectedGraph::Nout` map while multiple threads are working, a `unique_lock` is used on the mutex under a RAII scope to simplify resource managment [3]. This ensures that threads safely access the `DirectedGraph::Nout` map, one at a time, while allowing other threads to proceed with non-conflicting tasks without delays.

Further modifications were made on the components of the Vamana Algorithm to ensure the thread safety of the function. In this particular scenario, the synchronization between the `DirectedGraph::RobustPrune` and `DirectedGraph::GreedySearch` was achieved through condition variables under a readers-writers scenario [2]. Greedy Search takes the role of the reader as it is a read-only function, and requires mutual exclusion with any other writers, but not any other readers. In symmetry, the Robust Prune takes the role of the writer who requires exclusive entry among everyone, in order to enter its critical sections and modify the graph.

Modifications on the Robust Prune include the batch-adding of neighbors after keeping track of them, to reduce the frequency of entering its critical section, thus enabling other threads to obtain the locks more efficiently.

Finally the `DirectedGraph`'s `addEdge`, `removeEdge`, `clearNeighbors`, and `addBatchNeighbors` methods acquire `unique_lock`s under RAII scopes [3] only when required—if in a threaded environment—to ensure exclusive modifications on the `DirectedGraph::Nout` map.

Parallel Filtered Vamana

The parallel filtered Vamana unlike the unfiltered Vamana, does not require locks because categories are processed independently, ensuring there are no conflicts between threads. Each thread works on one category at a time, processing all nodes within that category without interfering with nodes from other categories. Before launching the threads, the algorithm sorts the categories based on their size to balance the workload more effectively across threads—the LPT Heuristic Optimization in Section 5.5.3. Additionally, the `DirectedGraph::Nout` map is pre-reserved with sufficient space to accommodate all edges, avoiding the need for resizing or rehashing during the algorithm.

Parallel Stitched Vamana

The parallel stitched Vamana algorithm does not require locks for category processing because each thread processes one category at a time, ensuring no conflicts between threads. Before launching the threads, the algorithm sorts the categories by their size to balance the workload more effectively and assigns larger categories to threads earlier—the LPT Heuristic Optimization in Section 5.5.3. A separate mutex is used once the subgraph is processed and its edges are stitched into the main graph.

Parallel Querying

The synchronization in this implementation uses the same approach as the parallel Rgraph. A mutex is used to ensure thread-safe access when assigning queries to threads. Once a thread retrieves its assigned query, it releases the lock, allowing other threads to proceed. There is no need for additional synchronization mechanisms, as each thread writes to a distinct index in the `returnVec`, ensuring thread safety without conflicts.

5.5 Heuristic Optimizations

5.5.1 Starting Point Selection

In the original paper introducing the Vamana Index, it was suggested to use the medoid of the dataset as the starting point for the greedy search algorithm [9]. Computing the medoid of a dataset, however, is a computationally expensive task. To mitigate this, we compute the

medoid only once during the Index Creation Phase and store it for subsequent use. In datasets with multiple categories, we calculate the medoid for each category—using the threshold t as described in section 4—and store these in a map. To approximate the total medoid of the entire dataset as if it were unfiltered, we select the medoid from the set of categorical medoids and store it as well.

As part of an optimization attempt, we explored the option of selecting a random point from either a specific category or the entire dataset, in order to bypass the computational cost of medoid calculation. However, this approach resulted in a significant decrease in the recall score, highlighting the importance of a carefully selected starting point.

Consequently, we chose to adhere to the original recommendation of computing the medoid, even at the expense of additional computational resources. This approach, while more costly, is executed only once during the Index Creation Phase and yields significantly higher recall scores compared to the random starting point approach.

5.5.2 Extra Random Edges for Unfiltered Queries

Since data points can belong to at most one category, the resulting index from either the filtered or stitched Vamana algorithms contains no edges between points from different categories [5]. Consequently, the choice of starting point directly impacts the available neighbors within its own category, as edges only exist between data points within the same category.

Initially, we sought to improve recall scores for unfiltered queries in a filtered index by **accumulating the results** from querying each category individually. However, this approach required performing as many queries as there were categories for each unfiltered query, which despite yielding extremely high recall scores, it was computationally inefficient and resulted in very long query times. To address this computational inefficiency, we propose a heuristic approach to replace the accumulation method.

This heuristic replaces the need for accumulating results by adding random edges from each vertex in the graph to any other vertex, regardless of the category it belongs to. The vision was to generate paths that lead to the true neighbors, while avoiding excessive overload on the query process. However, this strategy introduces additional overhead during the Index Creation Phase, as well as increases the overall size of the index. Despite these drawbacks, it significantly speeds up querying and maintains relatively high recall scores.

Initially, we hypothesized that the number of extra random edges should be close to the number of categories present in the dataset. However, this approach resulted in excessive computational stress and significantly increased the size of the index. Interestingly, even smaller numbers of random edges were found to improve the recall score. Due to the variability in the optimal number of extra random edges across different datasets, we will not present the results of our own sensitivity study. Instead, we recommend conducting a sensitivity analysis for each dataset to identify the best number of random edges that balances performance and efficiency.

5.5.3 LPT scheduling

Effective Load Balancing and job scheduling in multi-threaded environments is vital for optimal resource allocation. As described in section 5.4.2, the Filtered Vamana Algorithm and the Stitched Vamana Algorithm allocate a single thread per category in their parallel implementations.

In this context—per-category parallelization—a job's processing time can be estimated by the density of the specific category allocated to the worker thread. In section 4 we described the categorical mapping of the data that happens during the graph population. Leveraging information from this mapping we can greedily assume that the job's processing time is directly proportional on the number of nodes belonging in that specific category.

This heuristic optimization aims to reduce the total computational time by assigning larger jobs to the workers at first. By doing so, while a worker thread is tackling a large job, another worker with less workload will have finished and get assigned another, even shorter job. This ensures that no large job is left for last. By this assurance we gain the benefit of not having idle threads with no job left to be assigned while a single thread is working on a large problem on its own.

We implement a simplification of the Longest Processing-Time First heuristic (LPT) to greedily schedule the jobs by their descending workload. The simplification is as follows:

- No worker can have more than a single active job at any time, nor any other job scheduled for later. Jobs are obtained at a worker's availability.
- No two workers can work at the same job at the same time.
- Jobs are atomic. Once a job has been assigned to a worker, that specific worker alone will finish the job.

Under this scope it is intuitive that the simplified LPT heuristic will reduce wait times between the jobs, and also ensure that the threads getting assigned a job at a later time will not take more time to complete than the job assigned to the previous thread, especially if the large job would have been towards the end. This property stays true for any job in the LPT schedule, and therefore the exit time is expected to be shorter, as most workers will always have a job to do.

There exists a proven upper bound for the exit time of the non-simplified LPT, that says that it will never be more than $\frac{4}{3} \cdot \texttt{optimal}$. The proof can be found in *A Direct Proof of the 4/3 Bound of LPT Scheduling Rule* [13].

However, there exist cases where the Optimal Schedule is not good. One example could be degenerate distributions of data among categories, where a single category contains almost all the data points, whereas the rest of the categories contain only a single point. In such cases, the total time is bottle necked by the time it takes the massive job to finish, while at the same time starving the other threads.

A solution to this problem could be to assign multiple workers for the same job depending on the estimated workload. However, this is not always possible because of the limitations of which algorithms can be parallelized without compromising the integrity of the result.

5.6 Skipping R-Graph Initialization

In the DiskANN [9] paper it is suggested that the graph is initialized with R random outgoing edges from each vertex. However, in the Filtered DiskANN [5] this step is omitted. Upon receiving good results in the implementation of Filtered Indices, it was curious to see whether

good results could be achieved by skipping this step in the non-filtered Vamana Index. Consequently, by skipping this step in the Vamana Index, it means also skipping it in the Stitched Vamana, since it makes use of the already implemented algorithm. The results of these optimization show that skipping the initialization step comes at no cost, and also decreases the time required to build the index. More details can be found in Section 6.4.

5.7 Approximation of Medoids and memoization

Inspired by the algorithm for finding the medoids proposed for the Filtered Vamana Indices in the Filtered DiskANN paper [5], we implemented our own method that approximates the medoid.

Our own method relies on a `threshold` hyperparameter belonging in the range $(0, 1]$—a percentage. For a set S that we want to approximate its medoid, we calculate $\lceil t \cdot |S| \rceil$. This is the size of the *sampleSet*. The sampleSet will then be populated with that many samples from the original set $|S|$, avoiding re-sampling of the same node twice. The medoid is then calculated on the sampleSet, effectively reducing the workload by a fraction, equally for any set of any size.

In the case of an index with filtered data, the medoids are computed per-category and stored in a map for further use. In the case of unfiltered data, the single medoid is also stored in a different structure. In the case of filtered data but when performing an unfiltered query, a good starting point would be the medoid of all nodes in the graph. Therefore, the total medoid needs to be calculated.

Since the filtered medoids have already been calculated for each category before the querying phase, we can approximate the total medoid by finding the medoid of the approximate medoids. This final optimization step significantly reduces the workload of the approximation by leveraging previous stored knowledge.

The original authors support the use of medoid approximation methods for workload and time reduction, and our high recall scores further support that the approximation remains an acceptable optimization. For this reason, along with the additional delays, the explicit effect of this optimization will not be presented in contrast to the implementation without the medoid approximation.

5.8 Other possible optimizations that were not implemented

In this section, we will outline some possible optimizations that were not implemented, and the reason that we decided not to implement them.

5.8.1 Caching and Memoization for Frequently Calculated Distances

Caching and memoization were not implemented for the distance function due to space limitations. However, a potential improvement could involve the use of a limited-size cache with an appropriate replacement policy, such as Least Recently Used (LRU) or Most Recently Used (MRU) [6]. To determine the most suitable policy for this use case, a frequency analysis of distance computations could be conducted. Alternatively, different replacement policies could be applied and evaluated empirically to identify the approach that yields the best balance between performance and resource constraints.

5.8.2 Extra Pseudo-Random Edges

A potential refinement of the heuristic described in Section 5.6.2 could involve adding pseudo-random edges by limiting the number of additional edges within a specific category. This would help mitigate selection biases in dense categories, encouraging the formation of more diverse paths between underrepresented categories. The goal is for a more robust search process, which could further increase the recall score. However, this refinement was not implemented, as the random edge approach already yields satisfactory results without introducing excessive complexity.

5.8.3 Hybrid Scheduler for Filtered Vamana Indices

Upon completing the challenging task of synchronization of the components of the Vamana Indexing Algorithm [9], we allow for multiple workers to work at a single job. Under the scope of Filtered Indices, the Filtered Vamana Algorithm could be adapted in the same way as the Vamana Algorithm was adapted—described in detail in Section 5.4.2.

As discussed in section 5.6.3, the optimal schedule on a degenerate distribution is still not a good one, and a possible solution to this problem is dynamically assigning multiple workers on a job depending on the estimated workload.

A Hybrid Scheduler would use heuristics or an in-depth analysis of the distribution of the data in categories, and combine them with the workload estimation in order to dynamically determine the correct amount of workers to assign to a job, on the cost of the concurrency of running jobs.

While promising, dynamically offering a balance between concurrency and number of workers proved to be a complex task requiring further analysis of the data. Because our implementation aims to be generic and support any dataset, it is very difficult to design an effective scheduler that provides balanced results for many different distributions.

6 Benchmarks

To evaluate our implementation across different argument sets, we designed a simple benchmark test consisting of two distinct phases:

- **Index Creation Phase:** Constructs the Index based on the specified arguments.
- **Querying Phase:** Performs queries on the index using a predefined set of queries

6.1 Evaluation Metrics

The benchmark test gathers data during runtime and further processes it to compute the following evaluation metrics:

- **Index Creation Time:** Measures time (s) taken to build the index.
- **Index Querying Time:** Measures time (s) taken for querying the index. Categorized into:
  - Total Querying Time
  - Querying Time for Filtered Queries

  - Querying Time for Unfiltered Queries

- **Recall Score:** The percentage of correctly identified neighbors with no regard for position $\frac{|X \cap T|}{|X|}$
  - Total Average Recall Score
  - Average Recall Score for Filtered Queries
  - Average Recall Score for Unfiltered Queries

- **Queries Per Second (QPS):** The frequency of query performance. Categorized into:
  - Total QPS
  - Filtered QPS
  - Unfiltered QPS

- **Average Time for Single Query (ATSQ):** The Average Time (ms) it takes to complete an individual query. Categorized into:
  - Total ATSQ
  - Filtered ATSQ
  - Unfiltered ATSQ

- **Average Out Degree (AOD):** The average number of outgoing edges per vertex in the graph.

These metrics were selected to provide insights into various performance aspects, enabling optimizations aiming for higher accuracy, faster index creation, reduced querying time, minimal index size, or a balance between the previous objectives.

6.2 Benchmarks on dummy datasets

For a quicker evaluation of the argument sets, the Benchmark tests were conducted using reduced-size dummy datasets. The index was created on a dummy set of data points. The queries performed during the Querying Phase along with their corresponding ground truth for their recall score evaluation were all sourced from SIGMOD Contest [1].

The `--dummy` flag is used for setting the appropriate values on the corresponding arguments for the dataset, query, and ground truth selection, along with the required dimensions.

6.3 Argument Sets

The scientific method for experimentation dictates that to evaluate the effect of a specific parameter on the outcome, all other parameters must remain fixed [11]. During the development process, we came about an argument set which contains optimizations that achieves a highly-competitive, all-around balance across the optimization goals outlined in section 6.2. This argument set is referred to as the `optimal configuration`. However, it is important to emphasize that the term "optimal" is highly contextual, strictly dependent on the optimization objective.

In our experiments we perform an **ablation study** over the `optimal configuration` in order to assess the contribution of each isolated parameter. An ablation study involves selectively

stripping away components from a base configuration and measuring the impact on the final result [7]. By removing, replacing, or inserting an argument each time, variants of the `optimal` configuration arise. Each of the modified variants were tested under identical conditions, using the same benchmark tests and evaluation metrics, to ensure the comparability of results.

The variant argument sets are shown in Tables 2. Table 3 contains the explicit command line commands used to run each variation with the specific set of arguments.

Index Type	Configuration	Number of Threads	SIMD distance	No Rgraph	Priority Queue	Random Start	Extra Edges	Accumulate Unfiltered
Vamana	Optimal	26	✓	✓	✓		15	
Vamana	Serial	1	✓	✓	✓		15	
Vamana	Naive Euclidean	26		✓	✓		15	
Vamana	No Extra Edges	26	✓	✓	✓			
Vamana	GreedySearch using Set	26	✓	✓			15	
Vamana	Random Start	26	✓	✓	✓	✓	15	
Vamana	Initialized with Rgraph	26	✓		✓		15	
Index Type	**Configuration**	**Number of Threads**	**SIMD distance**	**No Rgraph**	**Priority Queue**	**Random Start**	**Extra Edges**	**Accumulate Unfiltered**
Filtered Vamana	Optimal	26	✓		✓		15	
Filtered Vamana	Serial	1	✓		✓		15	
Filtered Vamana	Naive Euclidean	26			✓		15	
Filtered Vamana	No Extra Edges	26	✓		✓			
Filtered Vamana	Accumulate Unfiltered	26	✓		✓			✓
Filtered Vamana	GreedySearch using Set	26	✓				15	
Filtered Vamana	Random Start	26	✓		✓	✓	15	
Index Type	**Configuration**	**Number of Threads**	**SIMD distance**	**No Rgraph**	**Priority Queue**	**Random Start**	**Extra Edges**	**Accumulate Unfiltered**
Stitched Vamana	Optimal	26	✓	✓	✓	✓	15	
Stitched Vamana	Serial	1	✓	✓	✓	✓	15	
Stitched Vamana	Naive Euclidean	26		✓	✓	✓	15	
Stitched Vamana	No Extra Edges	26	✓	✓	✓	✓		
Stitched Vamana	Accumulate Unfiltered	26	✓	✓	✓	✓		✓
Stitched Vamana	GreedySearch using Set	26	✓	✓		✓	15	
Stitched Vamana	Subgraph Random Start	26	✓	✓	✓		15	
Stitched Vamana	Subgraphs Initialized with Rgraph	26	✓		✓	✓		

Table 2: Argument Sets - Variant Configurations for all index types

Index Type	Configuration	Command Line Arguments
Vamana	Optimal	`./bin/main --vamana --stat --dummy --only_unfiltered -n_threads 26 -distance 1 --no_rgraph --pqueue -extra_edges 15`
Vamana	Serial	`./bin/main --vamana --stat --dummy --only_unfiltered -n_threads 1 -distance 1 --no_rgraph --pqueue -extra_edges 15`
Vamana	Naive Euclidean	`./bin/main --vamana --stat --dummy --only_unfiltered -n_threads 26 -distance 0 --no_rgraph --pqueue -extra_edges 15`
Vamana	No Extra Edges	`./bin/main --vamana --stat --dummy --only_unfiltered -n_threads 26 -distance 1 --no_rgraph --pqueue`
Vamana	GreedySearch using Set	`./bin/main --vamana --stat --dummy --only_unfiltered -n_threads 26 -distance 1 --no_rgraph -extra_edges 15`
Vamana	Random Start	`./bin/main --vamana --stat --dummy --only_unfiltered -n_threads 26 -distance 1 --random_start --no_rgraph --pqueue -extra_edges 15`
Vamana	Initialized with Rgraph	`./bin/main --vamana --stat --dummy --only_unfiltered -n_threads 26 -distance 1 --pqueue -extra_edges 15`
Index Type	**Configuration**	**Command Line Arguments**
Filtered Vamana	Optimal	`./bin/main --filtered --stat --dummy -n_threads 26 -distance 1 --pqueue -extra_edges 15`
Filtered Vamana	Serial	`./bin/main --filtered --stat --dummy -n_threads 1 -distance 1 --pqueue -extra_edges 15`
Filtered Vamana	Naive Euclidean	`./bin/main --filtered --stat --dummy -n_threads 26 -distance 0 --pqueue -extra_edges 15`
Filtered Vamana	No Extra Edges	`./bin/main --filtered --stat --dummy -n_threads 26 -distance 1 --pqueue`
Filtered Vamana	Accumulate Unfiltered	`./bin/main --filtered --stat --dummy -n_threads 26 -distance 1 --pqueue --acc_unfiltered`
Filtered Vamana	GreedySearch using Set	`./bin/main --filtered --stat --dummy -n_threads 26 -distance 1 -extra_edges 15`
Filtered Vamana	Random Start	`./bin/main --filtered --stat --dummy -n_threads 26 -distance 1 --random_start --pqueue -extra_edges 15`
Index Type	**Configuration**	**Command Line Arguments**
Stitched Vamana	Optimal	`./bin/main --stitched --stat --dummy -n_threads 26 -distance 1 --no_rgraph --pqueue -extra_edges 15`
Stitched Vamana	Serial	`./bin/main -stitched --stat --dummy -n_threads 1 -distance 1 --no_rgraph --pqueue -extra_edges 15`
Stitched Vamana	Naive Euclidean	`./bin/main --stitched --stat --dummy -n_threads 26 -distance 0 --no_rgraph --pqueue -extra_edges 15`
Stitched Vamana	No Extra Edges	`./bin/main --stitched --stat --dummy -n_threads 26 -distance 1 --no_rgraph --pqueue`
Stitched Vamana	Accumulate Unfiltered	`./bin/main --stitched --stat --dummy -n_threads 26 -distance 1 --no_rgraph --pqueue --acc_unfiltered`
Stitched Vamana	GreedySearch using Set	`./bin/main --stitched --stat --dummy -n_threads 26 -distance 1 --no_rgraph -extra_edges 15`
Stitched Vamana	Subgraph Random Start	`./bin/main --stitched --stat --dummy -n_threads 26 -distance 1 --random_start --no_rgraph --pqueue -extra_edges 15`
Stitched Vamana	Subgraphs Initialized with Rgraph	`./bin/main --stitched --stat --dummy -n_threads 26 -distance 1 --pqueue -extra_edges 15`

Table 3: Argument Sets - Variant Configurations for all index types as explicit Command Line commands

6.4 Benchmark Results on Dummy Datasets for each Argument Set

The results of the benchmark tests reveal the importance of each carefully chosen argument in the optimal configuration, over its ablated variants. The metrics outlined in section 6.1 highlight potential trade-offs between certain arguments.

Time for Index Creation and Querying

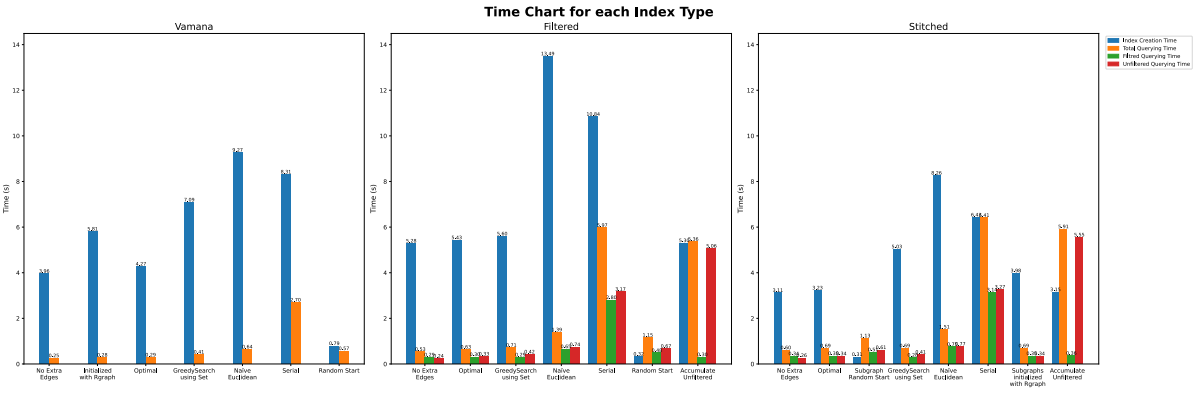

Figure 2: Time for Index Creation and Querying on Dummy Dataset

Recall Score

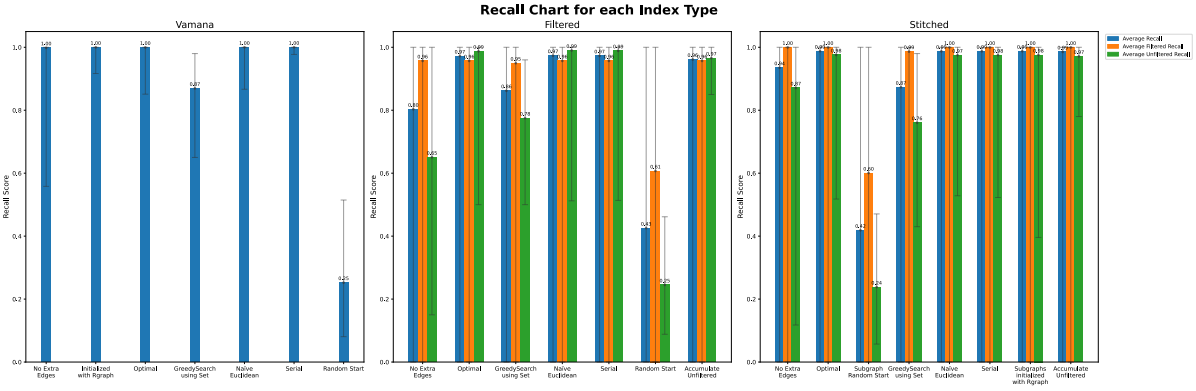

Figure 3: Recall Score for Dummy Dataset

Queries per Second (QPS)

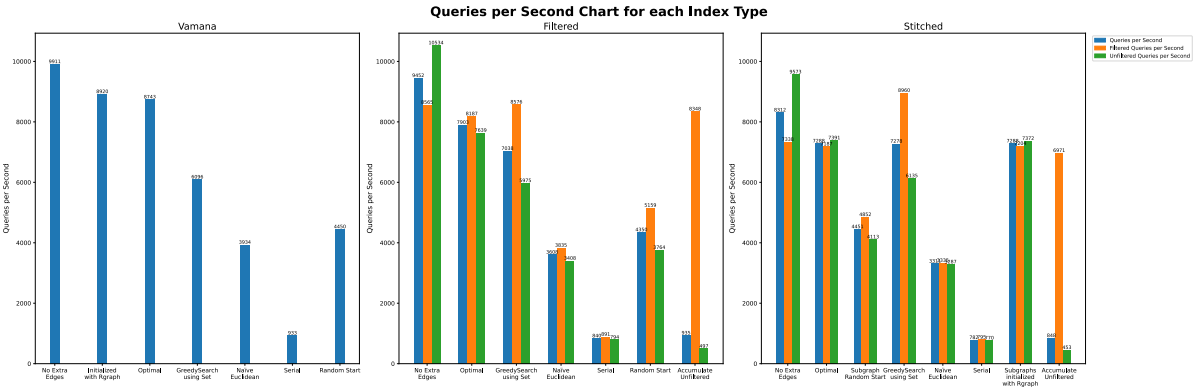

Figure 4: Queries per Second (QPS) for Dummy Dataset

Average Time for Single Query (ATSQ)

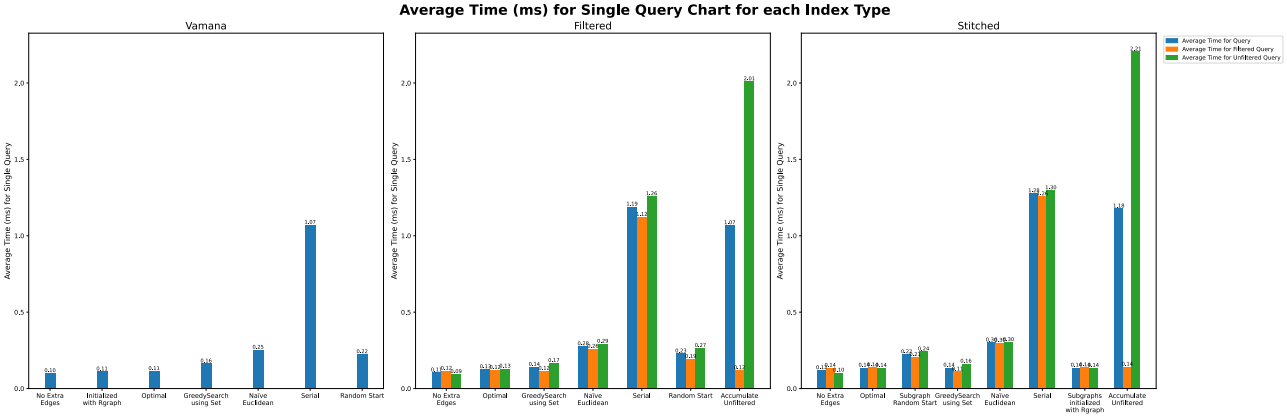

Figure 5: Average Time for Single Query for Dummy Dataset

Average Out-Degree (AOD)

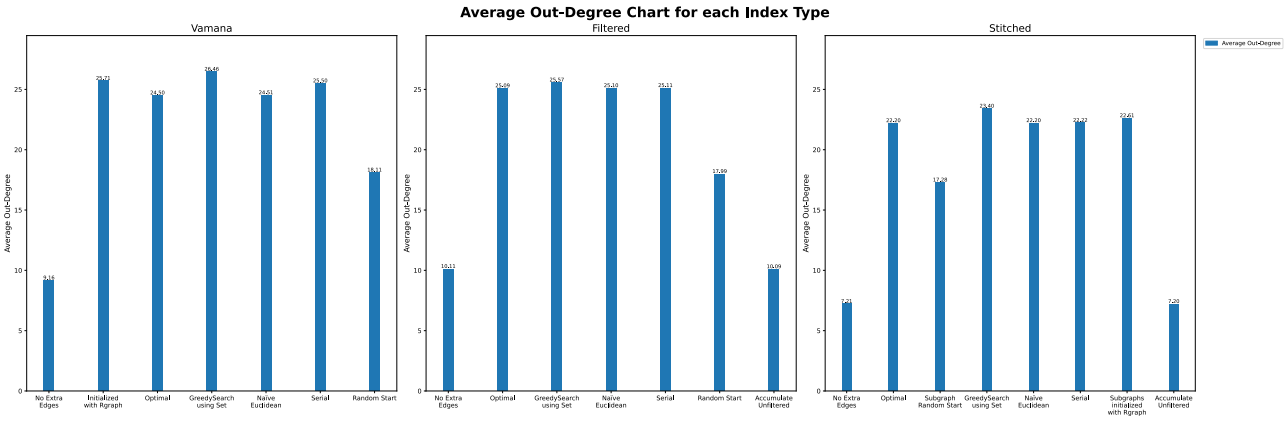

Figure 6: Average Out-Degree of Indices on Dummy Dataset

6.5 Benchmark Results on Million-Point Datasets

The Million-Point Dataset [1] consists of 1 000 000 data points in the same format as the dummy dataset. Due to its size, running various argument configurations proved challenging, as it consumed all available system resources and required hours to process, particularly during the development phase when the code was not yet optimized to reduce computation times. To address these limitations, we chose to run only the well-balanced optimal configuration described in the previous sections, with a single variation in the number of additional random edges. This approach aimed to improve the recall score for unfiltered queries while avoiding overloading the graph with excessive edges, which would increase its density disrupt the balance between our optimization objectives.

Time for Index Creation

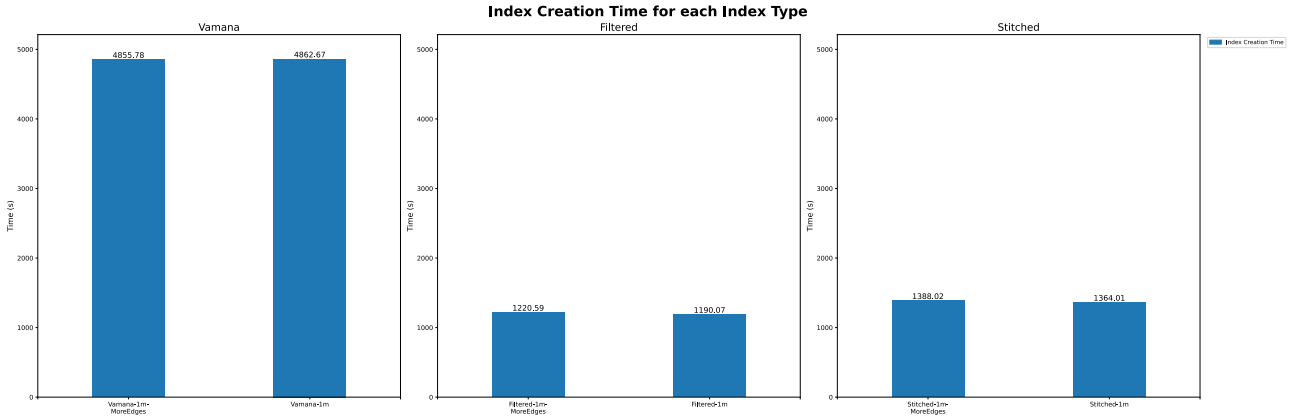

Figure 7: Time for Index Creation on Million-Point Dataset

Time for Index Querying

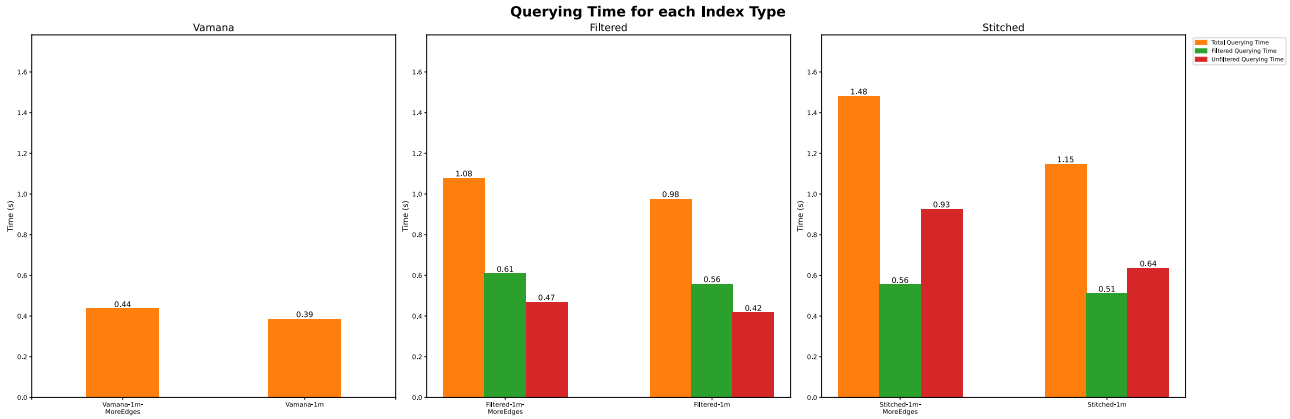

Figure 8: Time for Index Querying on Million-Point Dataset

Recall Score

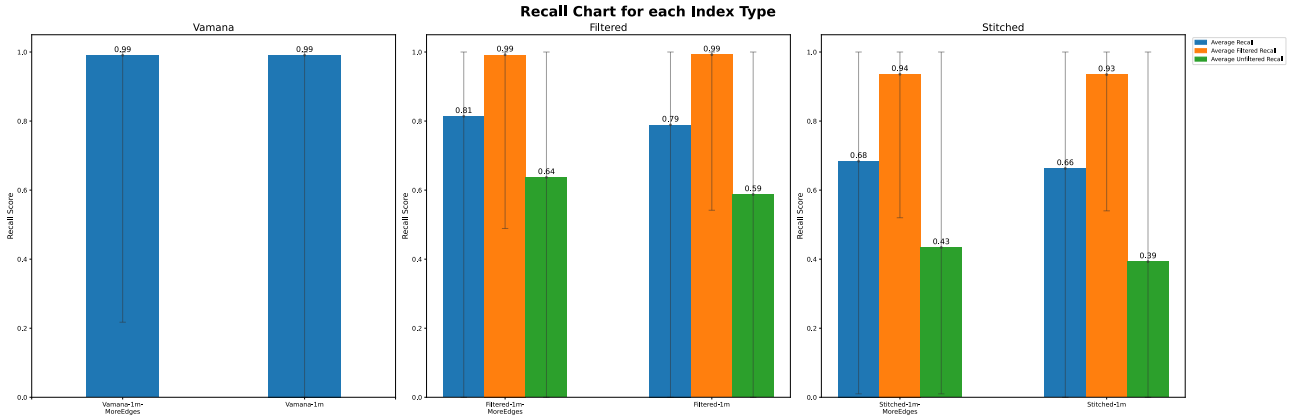

Figure 9: Recall Score for Million-Point Dataset

Queries per Second (QPS)

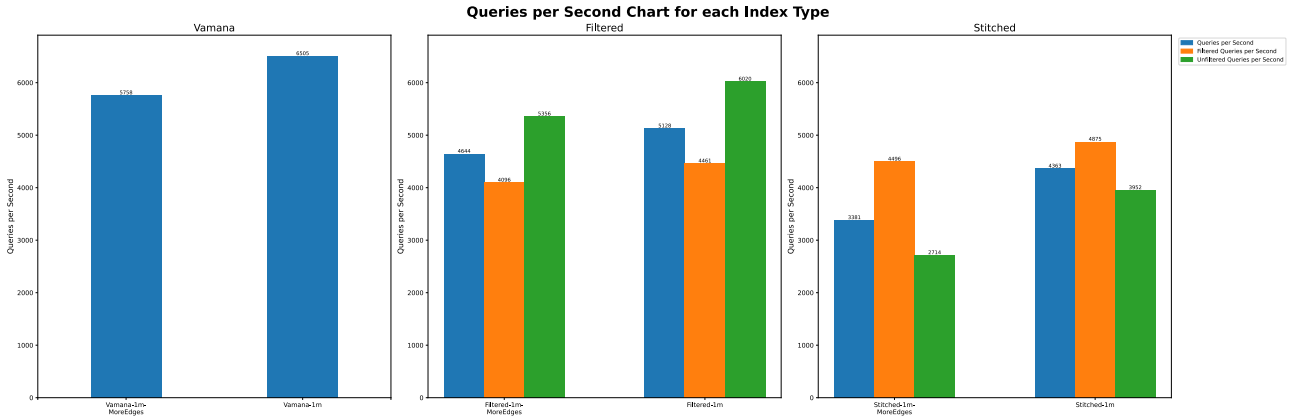

Figure 10: Queries per Second (QPS) for Million-Point Dataset

Average Time for Single Query (ATSQ)

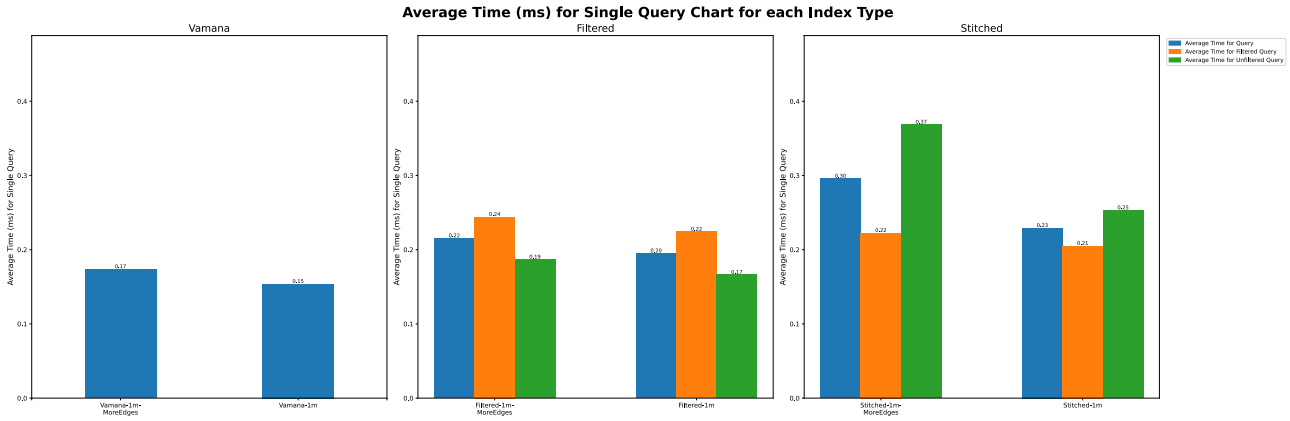

Figure 11: Average Time for Single Query for Million-Point Dataset

Average Out-Degree (AOD)

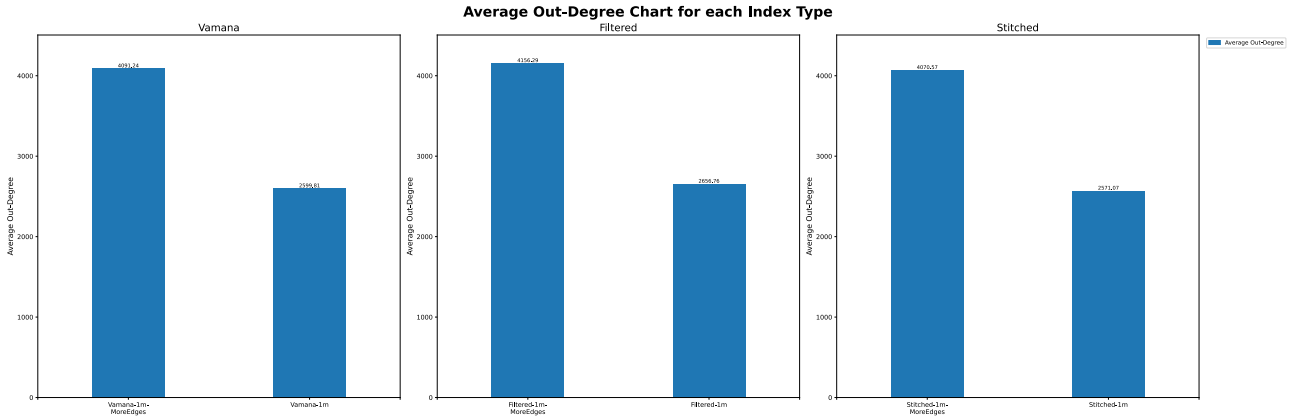

Figure 12: Average Out-Degree of Indices on Million-Point Dataset

6.6 Result Discussion

Our benchmarks aim to highlight the tradeoff between certain optimizations, each aiming for a specific objective, or a combination. The four objectives are

1. Index Creation Time Minimization
2. Querying Time Minimization
3. Recall Score Maximization
4. Graph Density Minimization

From the graphs we can see the following:

- **Utilizing threads** and implementing an **efficient distance function** proves critical, as it drastically reduces the required time for all operations, without negatively impacting any other optimization goals.
- **Using a Priority Queue** for the Greedy Search's candidate set $\mathcal{L}$ in contrast to an Unordered Set can slightly improve the time, and greatly improve the recall score.
- **Careful selection of the starting node** for the nearest neighbors search in Vamana-based Indices seems critical in achieving high accuracy. A robust choice for the starting node is the medoid.
- **Initializing the Index** before performing one of the three Vamana Algorithms is unnecessary, and can safely be skipped to reduce additional overhead in the Index Creation Phase.
- **Using robust methods to improve recall scores** for unfiltered queries on filtered indices is highly effective for achieving high recall but comes at a significant cost in reduced query frequency (QPS).
- **Performing unfiltered queries on filtered indices** without any method for improving the recall score yields abysmal results.
- **Using the extra edges heuristic** for improving the recall score for unfiltered queries on filtered indices can indeed significantly boosts the recall score, at a relatively low cost on querying time, but a rather greater cost at the graph's size and density.
- **The extra edges heuristic** does not scale well with larger datasets. It can overload the graph with redundant paths and consequently slow down query times. If required, alternative approaches should be considered.

The results highlight that not all optimizations are equally effective or scalable. Certain techniques proved highly effective towards specific goals but fell short on others. These findings emphasize the importance of carefully evaluating the trade-offs of each optimization in the context of the application's specific requirements and scalability challenges.

7 Conclusion

In this paper, we explored the impact of specific optimizations and modifications to three graph-based Vamana Indexing Algorithms for approximate K-Nearest Neighbor search in both filtered and unfiltered datasets. Our work demonstrates that the choice of optimizations directly influences performance trade-offs across four key objectives: Index Creation Time Minimization, Query Time Minimization, Recall Score Maximization, and Graph Density Minimization.

The experiments reveal that no single configuration universally excels across all objectives. Instead, the effectiveness of an optimization depends heavily on the target application and dataset characteristics. For instance, while multi-threading and efficient distance computations benefit all objectives by reducing computation times across all operations, methods such as the extra edges heuristic may improve recall at a small cost of querying time, but does not scale well for larger datasets. Similarly, prioritizing recall improvements for filtered queries may significantly impact query frequency (QPS), highlighting the need for balanced decisions based on application-specific requirements.

Our generic interface for interacting with Vamana-based algorithms ensures flexibility and compatibility across diverse data types. At the same time, targeted optimizations for common types such as floating point numbers, showcase how dataset-specific optimizations can achieve better results in practice.

Overall, this study emphasizes the importance of understanding the interactions between different optimization goals, and adapting configurations to meet the specific application's requirements. Future works could explore scalable alternatives to the extra edges heuristic that balance the recall improvement and graph density without complicating the querying process, as well as developing a general-purpose hybrid scheduler for varying distributions between categories.

References

[1] Blog, T. P. Sigmod contest 2024. https://transactional.blog/sigmod-contest/2024, 2024.

[2] Courtois, P. J., Heymans, F., and Parnas, D. L. Concurrent control with 'readers' and 'writers'. *Communications of the ACM 14*, 10 (1971), 667–668.

[3] cppreference contributors. Resource acquisition is initialization (raii). https://en.cppreference.com/w/cpp/language/raii, n.d.

[4] cppreference contributors. std::nth_element - c++ reference. https://en.cppreference.com/w/cpp/algorithm/nth_element, n.d.

[5] 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. Filtered-diskann: Graph algorithms for approximate nearest neighbor search with filters. In *Proceedings of the ACM Web Conference 2023 (WWW '23)* (New York, NY, USA, 2023), Association for Computing Machinery, pp. 3406–3416.

[6] Jain, A., and Lin, C. Cache replacement policies. *ACM Computing Surveys (CSUR) 51*, 2 (2018), 1–42.

[7] Meyes, R., Lu, M., Wermter, S., and Dangel, A. Ablation studies in artificial neural networks. *arXiv preprint arXiv:1901.08644* (2019).

[8] Mussabayev, R. Optimizing euclidean distance computation. *Mathematics 12*, 23 (2024), 3787.

[9] Subramanya, S. J., Devvrit, Kadekodi, R., Krishnaswamy, R., and Simhadri, H. V. Diskann: fast accurate billion-point nearest neighbor search on a single node. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems* (Red Hook, NY, USA, 2019), Curran Associates Inc., pp. 13766–13776. Article 1233.

[10] Taylor, R. N. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica 19* (1983), 57–84.

[11] Teodoro, G., Kurc, T., Taveira, L. F. R., Melo, A. C. M. A., Kong, J., and Saltz, J. Efficient methods and parallel execution for algorithm sensitivity analysis with parameter tuning on microscopy imaging datasets. *arXiv preprint arXiv:1612.03413* (2016).

[12] Wikibooks contributors. Algorithms/distance approximations, n.d.

[13] Xiao, X. A direct proof of the 4/3 bound of lpt scheduling rule.