NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS
SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

K23α: Software Development for Information Systems

Fall Semester 2024-2025

Efficient Approximate k-NN Search with Graph-Based Indexing and Filter Constraints

Michael-Raphael Kostagiannis
S.N.: 1115202100078

Giorgos Sofronas
S.N.: 1115202100180

Styliani Tziouma
S.N.: 1115202100184

Contents

1 Introduction

1.1 Problem Presentation

The task of approximating K Nearest Neighbors (KNN) in large-scale, high-dimensional data spaces represents a critical challenge in many real-world applications, including search engines, recommendation systems, and machine learning pipelines. The problem involves finding the closest K points to a given query point within a dataset of N points, where N can scale to billions.

In its exact form, KNN search suffers from the curse of dimensionality, making brute-force solutions computationally infeasible for high-dimensional and large-scale datasets. Consequently, Approximate Nearest Neighbor Search (ANNS) methods have emerged as practical alternatives, trading exactness for significant gains in efficiency and scalability.

Recent research efforts have introduced sophisticated indexing techniques to address these challenges. Notably, methods like DiskANN[1], which integrates graph-based indexing and disk-based storage, and Filtered-DiskANN[2], which adapts ANNS for filtered queries, have demonstrated state-of-the-art performance by optimizing trade-offs between memory, disk usage, and query throughput. These approaches leverage innovative graph structures, such as the Vamana algorithm for graph construction, to navigate large datasets effectively.

This report explores the development and evaluation of a C++ solution addressing a variation of the **SIGMOD 2024** Programming Contest, where each filtered query consists of a single categorical attribute. The solution focuses on being fast, accurate, and capable of handling large datasets, meeting the contest's challenging goals.

1.2 Hardware and System Specifications

Experiments were conducted on a system with the following hardware and software configuration:

- **CPU**
  - **Model:** 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz
  - **Cores:** 4 cores, 8 threads
  - **Cache:** 192 KiB L1d, 128 KiB L1i, 5 MiB L2, 8 MiB L3
- **Memory**: 16 GiB
- **Operating System**
  - **Distribution:** Ubuntu 24.04 LTS
  - **Kernel:** 6.8.0-51-generic
- **Compiler**: g++ 13.3.0

2 Previous Solutions

2.1 First Submission

The first submission was the initial logic implemented to compute k-nearest neighbors using a detailed and parameterized command-line interface. Users can provide inputs such as base and query vector files, ground truth data, vector dimensions, and search parameters like distance thresholds and list sizes through various flags. The Vectors class manages vector collections and caches Euclidean distances to expedite calculations. A Directed Graph class organizes neighbor relationships with unordered sets for efficient operations. Central to the implementation is the Vamana Indexing Algorithm, which builds and manages the k-NN graph. The program features menus for creating or loading graphs, finding neighbors, and saving structures, with robust unit tests and optimizations reducing execution times to as low as 25 seconds.

2.2 Second Submission

The second submission builds on the first implementation with notable enhancements in execution and algorithmic design. While the first solution focused on general k-NN computation, the second introduces Filtered Vamana and Stitched Vamana, achieving finer control with filter-based constraints. Sample executions show the filtered version reaching 99.70% recall in 27 seconds, while the stitched version improves to 99.96% recall in 31 seconds. These results surpass the previous implementation, demonstrating optimized performance and accuracy.

Key differences include:

- **Filter-based algorithms:** Filtered and Stitched Vamana extend the original Vamana by incorporating filtering constraints and subset-based graph construction.
- **Optimized Graph Stitching:** Introduces unionized edge sets to merge graphs efficiently.

Bash scripts also enable memory leak checks using Valgrind. Both executables are generated from a single main.cpp file using preprocessor directives, simplifying code management while maintaining flexibility.

3 Final Solution

Compilation and Execution

Along with the source and header files, found under `src/` and `include/` directories respectively, the project is instrumented using a group of three Makefiles:

1. Present under the `src/` directory, responsible for the compilation of the two main executables (`filtered` and `stitched`) and the `groundtruth` executable.
2. Can be found under the `tests/` directory. It is responsible for the compilation of the unit tests.
3. At the root of the project, from which both aforementioned makefiles are being controlled.

As far as users are concerned, **they should only use the makefile at the root of the project** to compile the executable programs and/or the unit tests.

The available commands are listed in the following table:

Compilation of FilteredVamana:	`make filtered`
Compilation of StitchedVamana:	`make stitched`
Compilation of groundtruth:	`make groundtruth`
Compilation of the unit tests:	`make tests`
Compilation of all of the above:	`make`
Cleanup command:	`make clean`

Object files are stored inside a real-time created directory called `build/`.

Each of the programs can be executed with a variety of flags, some of which are mandatory. For details, try to execute the wanted program, and a message would be printed, providing instructions. It should be mentioned that **all flags (both mandatory and optional) can be given in any order.**

Unit Testing

All of our functions and methods have been extensively tested, using acutest: a simple header-only C/C++ unit testing facility. The tests are focused on checking a wide range of both common and edge cases to ensure the program's reliability and results. All of our tests can be found under the `tests/` directory.

To simplify the tests' execution, a simple bash script (`scripts/run-tests.sh`) is provided, with which all the unit tests are executed at once. This script is also used as an automated testing tool in our GitHub repository, whenever a push or a pull-request occurs (GitHub Actions).

Scripts

A wide range of bash scripts is provided under the `scripts/` directory. More specifically, there are two types of scripts users should be concerned about:

1. `run-`*X*`.sh`: Executes *X*. For both vamana executables, this includes three different executions with different parameter options. For the unit tests, all tests are executed back-to-back.

2. `valgrind-`*X*`.sh`: Same instructions as the corresponding `run-`*X*`.sh` script, but with valgrind to ensure that there are no memory leaks.

***Note*: The scripts should be executed from the main/base directory of the project for the relative-paths inside them to be valid.**
Example: execution of filtered-vamana run: `./scripts/run-filtered.sh`

All other scripts (metrics-based) were just used for some of the data provided in this report.

3.1 Recommended Flags and Options

After thorough testing with numerable combinations, we have deducted that the most suitable parameters for the 10K dataset[3] and for our implementation are the following:

- FilteredVamana: a = 1.1, L = 150, R = 12, t = 5
- StitchedVamana: a = 1.1, L = 150, L_{small} = 100, R_{small} = 32, $R_{stitched}$ = 64, t = 5

3.2 Additional Flags

3.2.1 Initialize Graphs with Random Edges

In both the FilteredVamana and StitchedVamana algorithms, the output graph is first initialized to be an empty graph to which vertices and edges are progressively added. However, this implies that several disconnected subgraphs will be produced in the case of single-filtered datasets. By specifying the `--random-graph` flag during either program's execution, the graphs will be initialized with random edges between vertices in hopes of connecting the independent subgraphs and consequently increasing the performance on unfiltered queries.

3.2.2 Initialize Vamana-medoid Using a Random Point

The first step of the initial Vamana algorithm was to calculate the medoid of the whole dataset, which would be used as a starting point for all queries. However, because all Euclidean distances must be computed, determining the medoid becomes increasingly more difficult as the dataset grows larger. We can attempt to choose a random point as the dataset's medoid instead in order to minimize the graph's build time in StitchedVamana using the `--random-medoid` flag.

3.2.3 Initialize Vamana-medoid Using Medoid of a Random Subset

A possible intermediate approach between the medoid of the dataset and a random medoid would be to select a random subset of points and compute the medoid within this subset. This is done using the `--random-subset-medoid` flag.

3.2.4 Metrics

Here are the average metrics for each of the above optimizations. Metrics regarding time are counted in seconds, recall is counted as a percentage and the graph size is counted in bytes.

	Build Time	Filtered Time	Filtered Recall	Unfiltered Time	Unfiltered Recall	Total Recall	Query Time	Graph Size
No extra flags	2.45988	0.06965	99.72213	1.01343	99.68742	99.70470	1.08318	478235.00
--random-graph	3.52898	0.10733	8.41853	11.01787	98.41673	53.60622	11.12531	512543.60

Table 1: Filtered Vamana

	Build Time	Filtered Time	Filtered Recall	Unfiltered Time	Unfiltered Recall	Total Recall	Query Time	Graph Size
No extra flags	1.97201	0.10812	99.95992	1.44757	99.97171	99.96587	1.55578	684453.80
--random-graph	2.19599	0.22655	9.53166	28.28354	98.42935	54.16678	28.51020	757325.60
--random-medoid	1.68837	0.09862	99.96182	1.36412	99.97239	99.96716	1.46282	684759.60
--random-subset-medoid	1.76385	0.10260	99.95974	1.33405	99.97327	99.96656	1.43675	684836.00

Table 2: Stitched Vamana

- **--random-graph**: As expected, the build time and file size for both graphs have been increased. Unexpectedly, though, the recall for filtered queries has dropped to **less than one tenth** of the original recall! This occurs because each vertex of the graph is now randomly connected with other vertices without taking filters into consideration, leading to each point either having neighbors of different filters or little to no neighbors. Another thing worth noting is that the query time for unfiltered queries has significantly risen while simultaneously lowering their recall.
- **--random-medoid**: Indeed, both the build time and total query time have been reduced, and, surprisingly, the total recall for both filtered and unfiltered queries is practically the same.
- **--random-subset-medoid**: As we expected, this flag produces a graph with a build time that averages between dataset's original medoid and `--random-medoid`.

3.3 Limit Greedy Search Iterations

As you may have already noticed, the bottleneck of the program's execution is the time required to calculate the recall of unfiltered queries. In order to attain accurate queries, the FilteredGreedySearch function is invoked as many times as the number of medoids. Each medoid serves as the starting node for each respective search and the results are all placed in a vector which keeps the 100 nearest points. As a result, the unfiltered query time is **14.55 times slower** that the filtered query time in FilteredVamana and **13.39 times slower** in StitchedVamana! To overcome this, we can place a **limit** in the number of iterations of GreedySearch and FilteredGreedySearch using the `--limit <number>` flag.

3.3.1 Metrics

The following figures show how different metrics vary depending on the corresponding limit for both FilteredVamana and StitchedVamana.

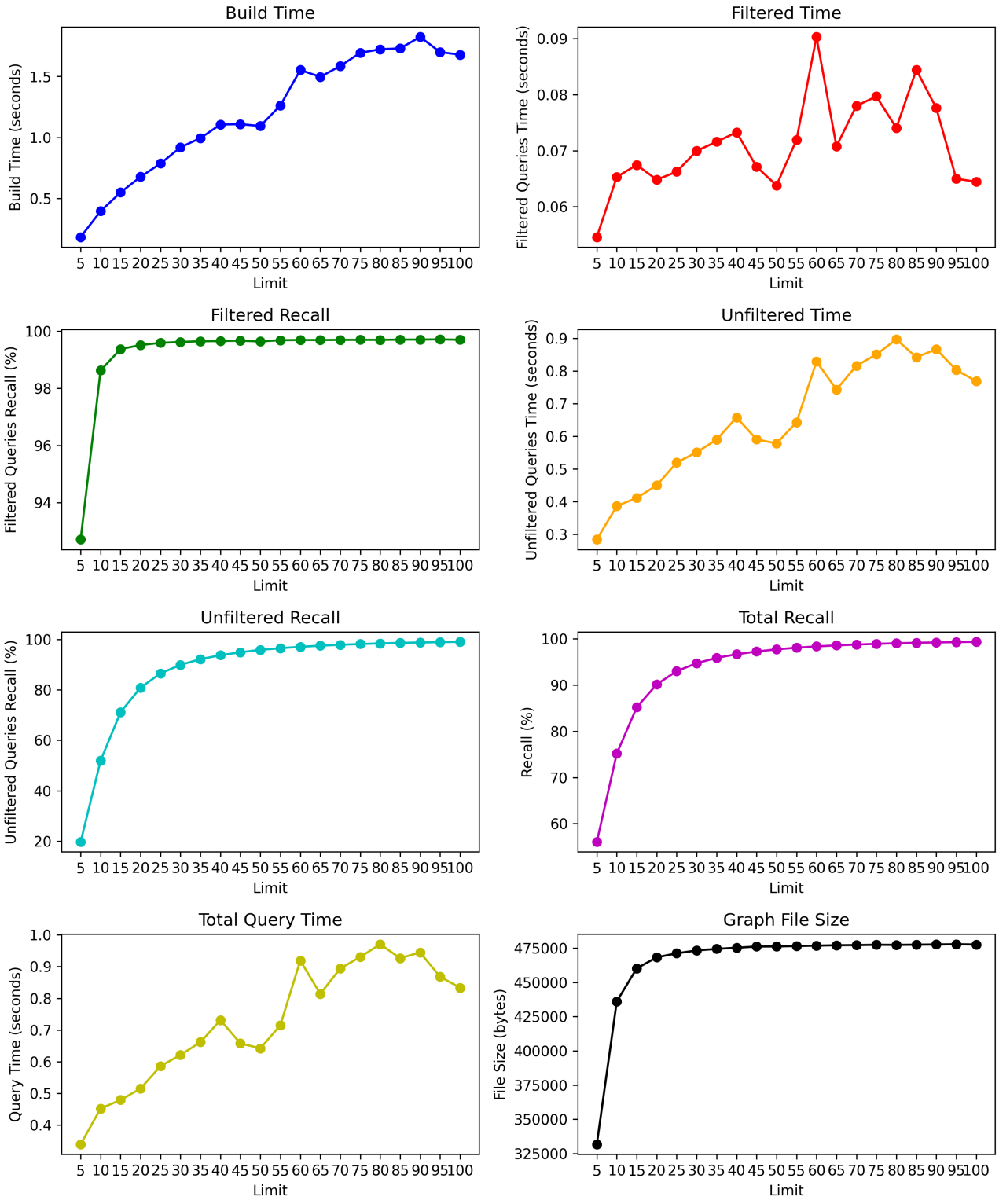

Figure 1: FilteredVamana --limit metrics

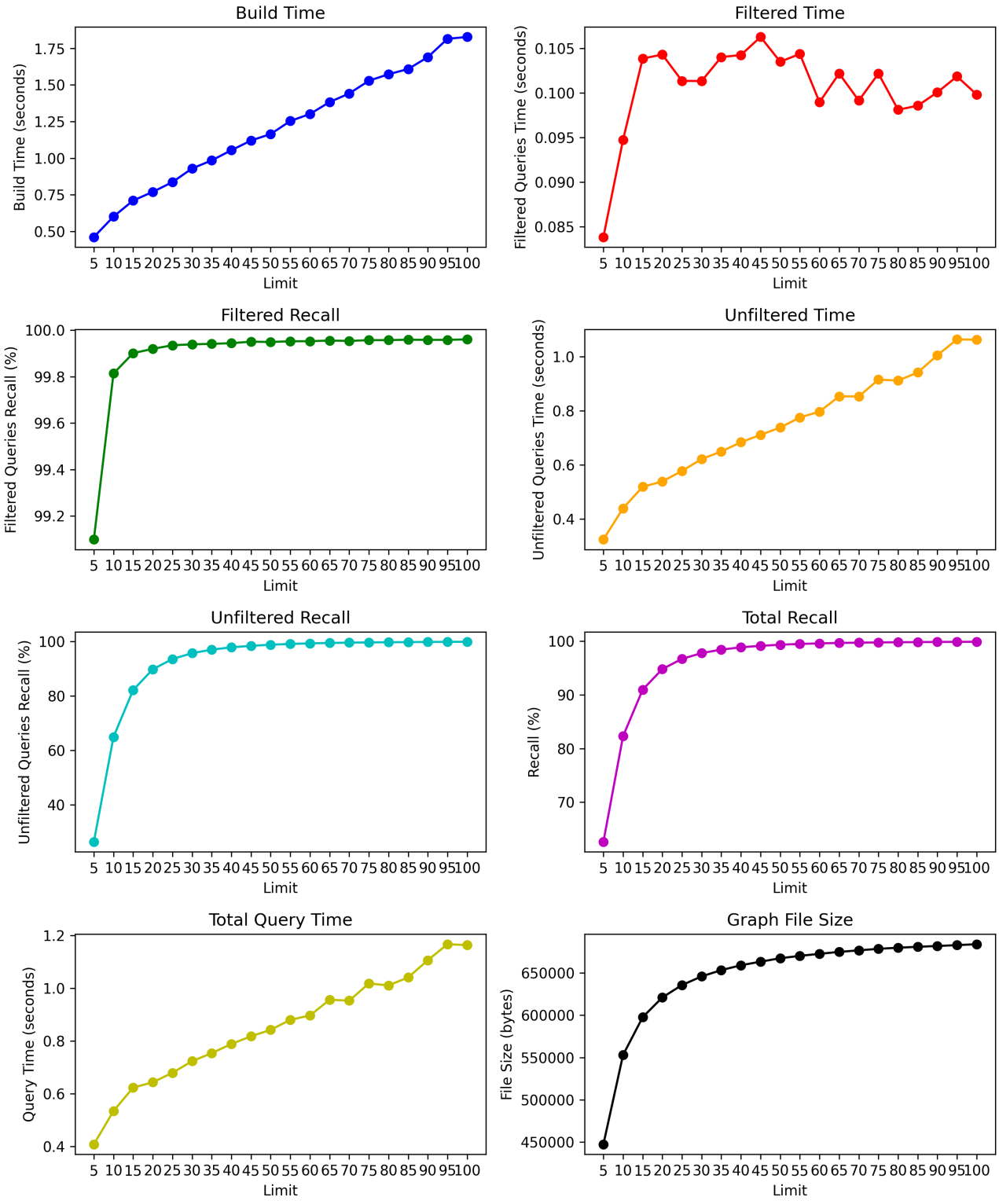

Figure 2: StitchedVamana --limit metrics

- While the build time, unfiltered queries time and total query time seem to be increasing **almost linearly**, all recalls and the average file size follow a **logarithmic** increase. This means that by setting a limit of 70 for FilteredVamana and 35 for StitchedVamana we can achieve a recall of about **95%** in around **half** the time!
- While the filtered query time graph might seem erratic or inconsistent, keep in mind that we are talking about a query time of **85-100 milliseconds**. The difference is almost non-existent.

3.4 Euclidean Distance Calculations Using SIMD

3.4.1 SIMD

SIMD (Single Instruction, Multiple Data) is a parallel computing paradigm that enables a single instruction to process multiple data points simultaneously. This approach is extensively used in modern processors to accelerate computational tasks such as image processing, machine learning, and scientific computations. By operating on data-vectors instead of individual elements, SIMD significantly enhances computational efficiency for data-parallel workloads.

3.4.2 Applying SIMD to Euclidean Distance Computation

The `euclidean_distance()` method is invoked numerous times throughout the program's execution, as indicated by our profiling results. Therefore, optimizing each call to this function has the potential to greatly improve the overall performance of the program. Observing the structure of Euclidean distance calculations, it is evident that they involve straightforward arithmetic operations (primarily subtraction and multiplication) performed on multi-variable data. This makes them an ideal candidate for leveraging SIMD instructions.

To achieve this optimization, we referred to the Intel Intrinsics Guide. The concept is straightforward: each vector register can store up to 256 bits. Considering that each floating-point number requires 4 bytes (32 bits), a single vector register can hold up to 8 floats. Using SIMD, we perform operations (like subtraction and multiplication) simultaneously, effectively processing 8-by-8 numbers in a single step. To handle cases where the vector's dimension is not perfectly divisible by 8, we ensure correctness by processing the remainder (a small number, between 0 and 7 elements) using a non-vectorized implementation.

3.5 Multithreaded Approach using OpenMP

In this section, we should discuss areas of code that were **parallelized**, using **multithreaded programming**.

3.5.1 OpenMP

OpenMP (Open Multi-Processing) is a widely used API that supports multi-platform shared-memory parallel programming. OpenMP simplifies the process of dividing tasks among multiple processors, making it ideal for improving performance in applications that require high computational power.

3.5.2 Queries calculation

This parallel implementation processes both filtered and unfiltered queries using **#pragma omp parallel for** with reduction clauses to ensure thread-safe aggregation of metrics like recall_sum, count, filtered_recall_sum, and filtered_count. For filtered queries, threads skip unfiltered queries and compute recall using calculate_filtered_recall, updating the shared metrics. Similarly, unfiltered queries are processed by skipping those with filters and computing recall via calculate_unfiltered_recall. Both sections leverage parallelism to maximize performance, with separate timers to measure execution time and calculate recall percentages for filtered and unfiltered queries.

3.5.3 Filtered graphs calculation in Stitched Vamana

This parallel implementation uses **#pragma omp parallel for** to construct and merge subgraphs for different filters in the dataset. Each thread processes a filter, extracting its associated points into an array (P_f) if the filter's size is greater than one. A directed subgraph (G_f) is then constructed using the vamana algorithm with the given parameters. To safely merge these subgraphs into the global graph (G), the critical section ensures thread-safe execution of the stitch operation.

3.5.4 Metrics

For the **FilteredVamana** version, the following table depicts the performance differences between the single and the multi threaded version, regarding the queries calculations:

	Build Time	**Filtered Time**	**Filtered Recall**	**Unfiltered Time**	**Unfiltered Recall**	**Total Recall**	**Query Time**
No Threads	2.79966	0.537548	99.7236	8.97923	99.6882	99.7065	9.51684
Threads (Queries calculation)	2.97067	0.103248	99.7186	1.6251	99.6824	99.7005	1.72872

Table 3: Single VS Multi Threaded Implementation in Queries-Calculation in FilteredVamana

For the **StitchedVamana** version, the following table shows the performance differences amongst the four possible versions: single-threaded, multithreaded in queries calculations, multithreaded in the *stitched* method, and multithread in both aforementioned areas:

	No Threads	**Threads in Queries Calculation**	**Threads *Stitched* Method**	**Threads in Both**
Build Time	3.55571	3.70862	2.43528	2.23939
Filtered Queries Time	0.950726	0.13797	0.693296	0.160785
Filtered Queries Recall	99.9631	99.9623	99.9615	99.9603
Unfiltered Queries Time	14.3176	2.32709	13.3043	2.33221
Unfiltered Queries Recall	99.967	99.9694	99.9714	99.969
Total Recall Percent	99.9652	99.9659	99.9665	99.9647
Total Query Time	15.2684	2.46515	13.9977	2.49336

Table 4: Single VS Multi Threaded Implementations in StitchedVamana

4 Ideas that didn't Enhance Performance

Here we present some optimization ideas that we assumed would improve the program's performance but were nonetheless slower. These ideas are not included in the final solution.

4.1 Parallelize Filtered Vamana

Since StitchedVamana was parallelized using OpenMP threads, one would assume that FilteredVamana could also be easily parallelized in order to increase performance. However, since FilteredVamana is split into multiple critical sections, it turns out that it becomes significantly slower during build and query time.

	Build Time	Filtered Time	Filtered Recall	Unfiltered Time	Unfiltered Recall	Total Recall	Query Time	Graph Size
Without threads	2.45988	0.06965	99.72213	1.01343	99.68742	99.70470	1.08318	478235.00
With threads	3.29167	0.07342	99.71108	1.22288	99.68190	99.69643	1.29639	478040.00

Table 5: Parallelizing Filtered Vamana

4.2 Comparing Euclidean Distances

Calculating the Euclidean distance between two points is an expensive operation, even if we include SIMD optimizations. We noticed that the final if statement of RobustPrune and FilteredRobustPrune has an extremely costly condition that requires us to determine the values of two distinct Euclidean distances. Theoretically, we could compute just the first one and utilize it as a cut-off threshold for computing the second, rather than computing both. In this manner, we could evaluate the condition before calculating the second Euclidean distance's real value and therefore save time. But in reality, it takes far longer than we anticipated to continuously check if the threshold was hit, nearly **tripling** the graph's build time!

4.3 Caching Euclidean Distances

A simple idea that was also used in the two previous submissions was to **cache** euclidean distances into a simple 2D array. This approach yielded excellent results in the previous submissions, due to the fact that cached euclidean distances would be instantly returned, without the need of recalculation.
Nevertheless, this approach is clearly **unsuitable for large datasets**, mainly due to the major **space complexity** overhead, and therefore it was rejected.

4.4 Storing Euclidean Distances in the Directed Graph

We explored the following approach: storing the euclidean distances directly on the directed graph during execution, rather than storing them in a separate array within the `Vectors` class. Whenever the insert method was called to add an edge to the graph, the Euclidean distance between the two vertices of the edge was also provided as an argument. As a result, for each vertex in the directed graph, the euclidean distances to its neighbors were cached/stored alongside the edges, and the neighbors were sorted based on their euclidean distances values. This optimization allowed the query calculations to retrieve precomputed euclidean distances from the directed graph, eliminating the need to recalculate them repeatedly.

This approach transformed the `unordered set` *(hash table)* of neighbors for each vertex into an `ordered set` *(red-black tree)*. Even though the idea was clever enough to boost performance in a theoretical level, the overhead and time difference that incurred due to the use of a less-performant data structure resulted in no performance increase. The total runtime remained unaffected, comparing it to the previously existing solution.

5 1M Dataset

We have also included useful metrics for the 1M dataset[4], which appears to be quite demanding. In order to keep execution time at a reasonable level, we have experimented with a plethora of parameter combinations. We have deduced that these are the most optimal parameters for each executable:

- `./filtered -b 1m/contest-data-release-1m.bin -q 1m/contest-queries-release-1m.bin -g 1m/contest-groundtruth-release-1m.bin -n 1000000 -m 5012 -a 1.1 -L 150 -R 12 -t 5 -i -1 --limit 60 -s filtered_1m_graph.bin`

- `./stitched -b 1m/contest-data-release-1m.bin -q 1m/contest-queries-release-1m.bin -g 1m/contest-groundtruth-release-1m.bin -n 1000000 -m 5012 -a 1.1 -L 150 -l 100 -r 32 -R 64 -t 5 -i -1 --random-medoid --limit 40 -s stitched_1m_graph.bin`

	Build Time	Filtered Time	Filtered Recall	Unfiltered Time	Unfiltered Recall	Total Recall	Query Time	Graph Size
FilteredVamana	771.434	0.65301	94.5409	343.77	90.9818	92.7542	344.423	50091940.00
StitchedVamana	547.855	1.84407	98.9367	344.592	95.4297	97.1762	346.436	84103320.00

Table 6: 1M Dataset Metrics

References

[1] Suhas Jayaram Subramanya, Devvrit, Rohan Kadekodi, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. *DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node*. In *Proceedings of the 33rd Conference on Neural Information Processing Systems (NeurIPS)*, 2019. URL: https://dl.acm.org/doi/pdf/10.5555/3454287.3455520.

[2] Siddharth Gollapudi, Neel Karia, Nikit Begwani, Swapnil Raz, Varun Sivashankar, Ravishankar Krishnaswamy, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premukumar Srinivasan, Amit Singh, and Harsha Vardhan Simhadri. *Filtered-DiskANN: Graph Algorithms for Approximate Nearest Neighbor Search with Filters*. In *Proceedings of the ACM Web Conference 2023 (WWW '23)*, 2023. URL: https://dl.acm.org/doi/pdf/10.1145/3543507.3583552.

[3] 10K dataset: https://github.com/transactionalblog/sigmod-contest-2024

[4] 1M dataset: https://zenodo.org/records/13998879