Software Development for Information Systems - K23α

Plain, Filtered and Stitched Vamana Algorithms

Final Report

Athens, January 2025

Team

#	Name	Email	UUID	Github
1	Christos Kaltsas	sdi2000289@di.uoa.gr	1115202000289	xcalts
2	Natalia Krikelli	sdi2000104@di.uoa.gr	1115202000104	nataliakrik

---

Github

di-uoa-project-k23a

Contents

Introduction

General

This project focuses on the development of software for information systems, with a particular emphasis on addressing the Approximate Nearest Neighbors (ANN) problem. Specifically, it implements the Vamana Indexing Algorithm (VIA), which is a modern method for efficiently solving the ANN problem in high-dimensional data.

The goal of this project is to create an efficient and flexible system that enables rapid search and retrieval of information in large databases, leveraging the capabilities of the vamana indexing algorithms. This implementation aims to improve performance and accuracy in applications requiring fast and reliable nearest neighbor searches.

Objectives

The main objectives of the project include:

- **Implementation of the Vamana Algorithm**: Development of a complete and functional version of the VIA tailored to the requirements of the ANN problem.
- **Performance Optimization**: Application of techniques to reduce execution time and resource requirements, ensuring the system's efficiency.
- **Accuracy Evaluation**: Conducting experiments to assess the accuracy and effectiveness of the implemented algorithm compared to existing methods.

Workflow

Command Parsing and Validation

The application begins by parsing and validating the command-line arguments. The main steps include:

1. Extracting the command, YAML configuration file path (--*conf*), and algorithm type (--*algo*) from the command-line input using the argh library.

2. Verifying the presence of the required arguments. If insufficient arguments are provided, the application throws an exception with a usage message.

3. Validating that the specified YAML configuration file exists and that the provided command is valid by calling *validateFileExists()* and *validateCommand()*.

4. Logging all parsed arguments and configurations for debugging purposes.

Dataset Parsing

The application parses datasets based on the specified configuration. This step includes:

1. Reading the data points from files (e.g., *.fvecs* or custom file formats) using helper functions like *parsePointsFvecsFile()*, *parseDummyData()*, or *parseDummyQueries()*.

2. For evaluation commands, parsing additional datasets such as query data and ground truth data using *parseQueriesFvecsFile()* and *parseIvecsFile()*.

3. Ensuring all required files exist and their contents are correctly formatted. Errors are thrown if any required files are missing or invalid.

4. Logging dataset sizes and sources for debugging.

Initialization

During initialization, the application performs the following:

1. Validates critical parameters from the YAML configuration file, such as *k*, *L*, and algorithm-specific values (e.g., *a*, *R*, *tau*). For example, ensures L > k to maintain the algorithm's validity.

2. Loads data points and initializes the chosen algorithm:

Vamana
Initializes the Vamana graph structure and performs indexing using the *index()* method.

Filtered Vamana
Loads data with dummy filtering and indexes points using filtering criteria.

Stitched Vamana
Initializes a stitched graph structure and indexes points with specific stitching parameters.

3. Saves the initialized graph to the specified file for reuse in evaluation.

4. Logs key initialization statistics, including graph construction times and parameter values.

Evaluation

The evaluation phase assesses the performance of the selected algorithm. Key steps include:
1. Loading the pre-constructed graph and ground truth data for comparison.

2. Running the algorithm's query processing:

Vamana and Filtered Vamana
Executes either filtered or unfiltered nearest-neighbor searches based on query type.

Brute Force
Provides true nearest neighbors as a baseline for recall computation.

3. Measures metrics such as recall and query processing time:
- Recall is calculated as the overlap between algorithm-generated nearest neighbors and the ground truth.
- Query times are aggregated to compute average times for filtered and unfiltered queries.

4. Logs progress using a *BlockProgressBar* and outputs recall and query timing results.

Statistics

At the end of execution, the application outputs detailed statistics for analysis:

1. Recall Metrics:
For both filtered and unfiltered queries, recall is reported as a percentage of correct matches out of total nearest neighbors.

2. Timing Metrics:
Average query time for both filtered and unfiltered queries is calculated and logged.

3. Graph Construction Metrics:
Logs construction times (e.g., medoid calculation time, graph indexing time) during initialization.

These statistics are displayed in the console and provide insights into algorithm performance, enabling comparison between configurations and algorithms.

To extend the analysis further, a custom Python script was developed to automate the execution of the program across a wide range of parameter configurations. This script systematically runs the program with varying values of key parameters, such as k, L, and a, and collects the resulting statistics for each configuration. Using the data collected, the script generates performance graphs to visualize relationships, such as recall versus query time for filtered and unfiltered queries, construction time versus parameter values, and comparative performance across algorithms like Vamana, Filtered Vamana, and Stitched Vamana.

These visualizations provide a comprehensive understanding of the algorithms' behavior under different conditions and are instrumental in identifying trends, trade-offs, and opportunities for optimization. Together, the program's built-in metrics and the custom Python-generated graphs create a robust foundation for evaluating and fine-tuning the algorithms for optimal performance.

Remarks

Vamana

As *L* and *R* increase in size so does the indexing time

As the graph expands and becomes denser with increasing values of L and R, it is logical to expect a corresponding increase in indexing time.

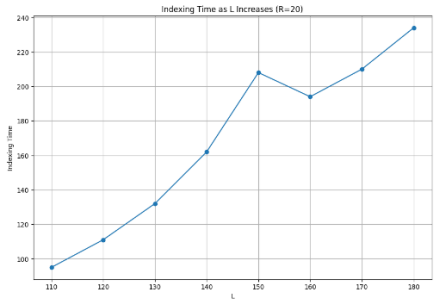

Figure 1- Indexing Time as L Increases (R = 20)

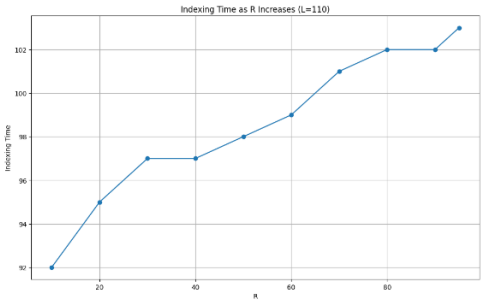

Figure 2 - Indexing Time as R Increases (R = 20)

As *L* and *R* increase, only *L* seems to affect the recall of the algorithm

The recall appears to increase linearly with the *L* parameter, eventually plateauing at 100%. In contrast, we observed no noticeable effect with changes to the *R* parameter.

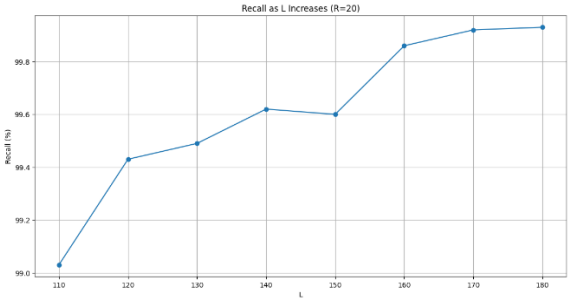

Figure 3 - Recall as L Increases (R = 20)

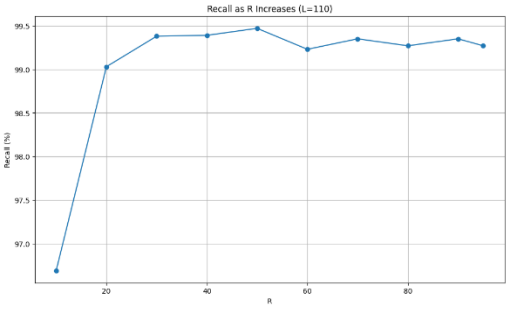

Figure 4 - Recall as R Increases (L = 100)

For small datasets, Vamana is slower than the Brute Force method

This performance gap diminishes as dataset size increases, where Vamana's optimized search capabilities become more apparent.

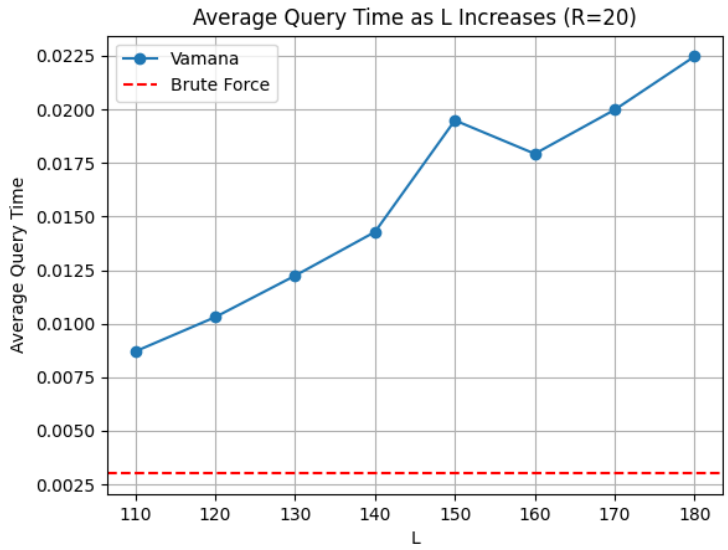

Figure 5 - Average Query Time as L Increases in Comparison to Brute Force Method

Filtered Vamana

Its indexing time is considerably faster than the Vamana one

For a dataset of the same size, we recorded the following times:

Algorithm	Dataset Size	Indexing Time
Vamana	10,000	~ 170 seconds
Filtered Vamana	10,000	~ 50 seconds

Figure 6 - Comparison between Vamana and Filtered Vamana Indexing Times

Filtered Vamana's average query time is faster than the Vamana algorithm

The Filtered Vamana algorithm is faster than the Brute Force method. For example, when *L*: 120 and *R*: 30, we got the following results:

Algorithm	Dataset Size	Average Query Time
Vamana	10,000	~ 0.012 seconds
Filtered Vamana	10,000	~ 0.006 seconds

Figure 7 - Comparison between Vamana and Filtered Vamana Indexing Times

Optimizations

Our team optimized the calculation of the medoid using threads. This optimization yielded the following results:

Description	Before	After	Change %
Parallelization of medoid calculation using threads	40 sec	7 sec	82.5%

Figure 8 - Optimizations along with the performance benefits

Conclusion

In conclusion, this project highlights the critical role of understanding the data we analyse in optimizing the performance and efficiency of algorithms like Vamana and Filtered Vamana. A deep comprehension of the dataset characteristics allows us to design and apply effective filters and select the most appropriate parameters, significantly enhancing both memory usage and algorithmic performance. By tailoring the filtering criteria and parameter configurations to align with the specific nature of the data, we can achieve more targeted searches, faster query responses, and reduced computational overhead. This underscores the importance of data-driven optimization strategies in building scalable and efficient systems, particularly when dealing with large and complex datasets.

References

External Libraries

Github | argh - A minimalist argument handler
Github | acutest - Simple header-only C/C++ unit testing facility
Github | rapidyaml - Parse and emit YAML, and do it fast
Github | indicators - Activity Indicators for Modern C++
Github | termcolor - ANSI color formatting for output in terminal

Papers

DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node
Filtered-DiskANN: Graph Algorithms for Approximate Nearest Neighbor Search with Filters