Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα

Ιανουάριος 2025

Μαυραΐδης-Κούβελας Κωνσταντίνος sdi0700101@di.uoa.gr - 1115200700101
Καραγιάννη Θωμάς sdi1800064@di.uoa.gr - 1115201800064

Περιεχόμενα

Introduction

K-Nearest Neighbors (KNN) algorithms are a type of proximity search algorithm used in various fields, including computer science, statistics, and data mining. The primary goal of KNN algorithms is to find the k most similar data points to a query point in a dataset. These algorithms are widely used in applications such as recommendation systems, image and video retrieval, and natural language processing.

Basic Vamana Algorithm

Vamana is a type of KNN algorithm that uses a combination of indexing and filtering techniques to efficiently search for nearest neighbors. The basic Vamana algorithm works by first indexing the dataset using a hierarchical data structure, such as a tree or a graph. This indexing step allows for fast pruning of the search space, reducing the number of candidate points that need to be evaluated.

Once the indexing step is complete, the algorithm uses a filtering technique to quickly eliminate points that are unlikely to be among the k nearest neighbors. This filtering step is typically based on a distance metric, such as Euclidean distance or cosine similarity.

The advantages of the basic Vamana algorithm include:

Efficient search: Vamana's indexing and filtering techniques enable fast search times, even for large datasets.
Scalability: Vamana can handle high-dimensional data and large numbers of data points.
Flexibility: Vamana can be used with various distance metrics and can be adapted to different types of data.
Filtered Vamana and Stitched Vamana

I'd be happy to provide a brief summary of the Filtered Vamana and Stitched Vamana algorithms. However, I want to clarify that I don't have specific information about the implementation of these algorithms in your workspace. If you could provide more context or information about the specific implementation, I'd be happy to try and provide a more detailed summary.

That being said, here is a brief summary of the Filtered Vamana and Stitched Vamana algorithms:

Filtered Vamana: This algorithm is an extension of the basic Vamana algorithm that uses additional filtering techniques to further reduce the number of candidate points. The filtering step is typically based on a combination of distance metrics and other features of the data.
Stitched Vamana: This algorithm is another extension of the basic Vamana algorithm that uses a stitching technique to combine the results of multiple Vamana searches. This technique is useful when the dataset is too large to fit into memory or when the search needs to be parallelized.

Filtered Vamana

Our implementation of the Filtered Vamana algorithm takes in several parameters, including the dataset, the parameter controlling the size of the visited list in the greedy search (L), the pruning parameter (a), and the maximum number of neighbors allowed for a point after pruning (R).

From our implementation, it appears that the Filtered Vamana algorithm works as follows:

Initialize the graph: The algorithm begins by initializing the graph using a graph initialization function.
Calculate the medoid: The algorithm then calculates the medoid of the graph using a medoid calculation function.
Traverse the graph: The algorithm traverses the graph in a random way without repetitions using a graph traversal function.
Run the greedy search: For each random point, the algorithm runs the greedy search algorithm to get the visited list V.
Run the robust pruning: The algorithm then runs the robust pruning algorithm to prune the neighbors of each point.
Filter the medoids: Finally, the algorithm filters the medoids using a filtering function.
The filtering step in the Filtered Vamana algorithm is based on a filtering structure that is used to further reduce the number of candidate points based on additional criteria. This filtering step is used to refine the results of the algorithm and improve its accuracy.

Overall, our implementation of the Filtered Vamana algorithm is a highly efficient and effective approach to indexing large datasets. By using a combination of graph initialization, medoid calculation, graph traversal, greedy search, robust pruning, and filtering, we can significantly improve the accuracy and efficiency of the algorithm.

Stitched Vamana

Our implementation of the Stitched Vamana algorithm takes in several parameters, including the dataset, the parameter controlling the size of the visited list in the greedy search (L_small), the pruning parameter (a), and the maximum number of neighbors allowed for a point after pruning (R_small).

From our implementation, it appears that the Stitched Vamana algorithm works as follows:

Divide the dataset into multiple sub-datasets, each corresponding to a filter.
For each sub-dataset, run the Vamana algorithm to get the indexed points.
For each indexed point, run the filteredRobustPrune algorithm to prune the neighbors.
Combine the results of each sub-dataset to get the final indexed points.
Our implementation uses a parallelized approach to speed up the computation. Multiple threads are created, each responsible for processing a subset of the filters. Each thread runs the Vamana algorithm on its assigned subset of filters and writes the results to a result file. The main thread then combines the results from each thread to get the final indexed points.

The Stitched Vamana algorithm is designed to take advantage of multi-core processors and large datasets. By dividing the computation across multiple threads, it can significantly speed up the indexing process

General Optimizations

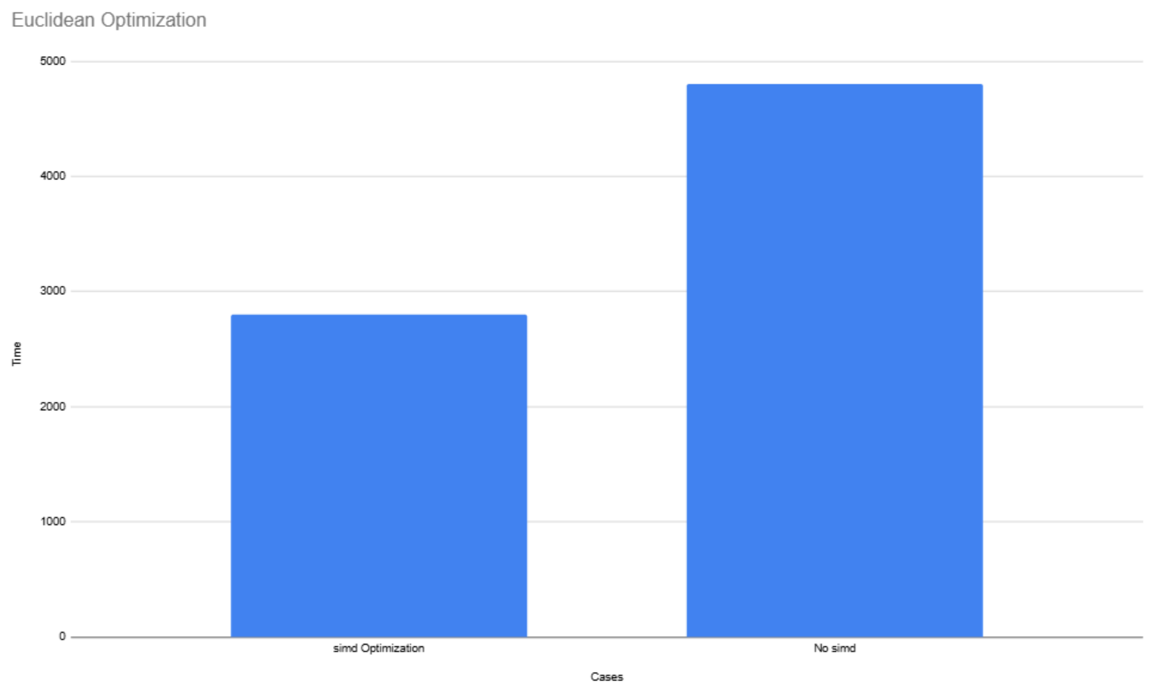
Optimizations in Squared Euclidean Distance Calculation

One of the key optimizations in our implementation is the use of SIMD (Single Instruction, Multiple Data) instructions to accelerate the calculation of squared Euclidean distances. Specifically, we utilized the 64-bit SIMD instructions to parallelize the computation, resulting in a significant reduction in processing time.

Advantages

The use of SIMD instructions in the squared Euclidean distance calculation offers several advantages:

Improved performance: By processing multiple data points simultaneously, SIMD instructions can significantly reduce the processing time. In our case, the optimization reduced the processing time for two test points of 100M dimensions from 0.48 seconds to 0.21 seconds, a speedup of approximately 56%.



Increased throughput: SIMD instructions can handle multiple data points in a single clock cycle, increasing the overall throughput of the system.

Better utilization of CPU resources: By leveraging the SIMD capabilities of modern CPUs, our implementation can make better use of available CPU resources, reducing the overall processing time.

Conclusion

The use of SIMD instructions in the squared Euclidean distance calculation is a key optimization in our implementation, offering significant performance improvements and increased throughput. While there are some potential disadvantages to consider, the benefits of SIMD instructions make them a valuable tool in the development of high-performance algorithms.

Pre-Calculating and Storing Distances

Another optimization we considered was pre-calculating and storing the distances between all points in the dataset. This approach would involve computing the distance between every pair of points and storing the results in a matrix or other data structure.

Time Complexity

However, this approach would have a time complexity of O(n^2), where n is the number of points in the dataset. This is because we would need to compute the distance between every pair of points, resulting in a quadratic number of computations. For large datasets, this would be prohibitively expensive in terms of computation time.

Memory Requirements

Furthermore, storing the pre-computed distances would require a significant amount of memory. The distance matrix would have a size of n x n, where n is the number of points in the dataset. For large datasets, this would require a substantial amount of memory, potentially exceeding the available memory on the system.

Conclusion

While pre-calculating and storing distances may seem like an attractive optimization, it is not feasible due to the high time complexity and memory requirements.

General Quality of Life Improvements

In addition to the performance optimizations discussed earlier, we also implemented several general quality of life improvements to enhance the overall efficiency and maintainability of the code.

Reuse of Memory

One such improvement is the reuse of memory through the use of pointers to vectors to read the base dataset to enable reuse of memory. By doing so, we avoid the need for additional memory allocations and deallocations, reducing memory fragmentation and improving overall memory efficiency. This approach also simplifies the code and reduces the risk of memory-related bugs.

We reuse the memory allocated for the vectors when reading the dataset for the graphs. By reusing the existing memory, we avoid the need for additional memory allocations and deallocations, reducing memory fragmentation and improving overall memory efficiency. This approach also simplifies the code and reduces the risk of memory-related bugs.

Improved Complexity in Medoid Calculation

Furthermore, we improved the complexity of the medoid calculation algorithm. By optimizing the algorithm, we reduced the computational complexity from O(n^2) to O(n), resulting in significant performance improvements. This improvement also simplifies the code and reduces the risk of errors.

Conclusion

These general quality of life improvements have significantly enhanced the overall efficiency and maintainability of the code. By reusing memory and improving the complexity of the medoid calculation algorithm, we have reduced memory fragmentation, improved performance, and simplified the code. These improvements have made the code more robust, efficient, and easier to maintain, and have paved the way for further optimizations and improvements

Filtered Vamana

All the result sets have used the 1m contest dataset.

PC used for filtered_vamana test.

- **Processor**: AMD Ryzen™ 7 8845HS with 8 cores, delivering multi-threaded performance.
- **Memory**: 16 GB of RAM, suitable for memory-intensive tasks.
- **Graphics**:
  - Integrated: AMD Radeon™ 780M Graphics.
  - Dedicated: NVIDIA GeForce RTX™ 4050 Laptop GPU for enhanced GPU-accelerated performance.
- **Storage**: 1 TB of disk capacity, providing ample space for data storage.
- **Operating System**: Ubuntu 24.04 LTS (64-bit), ensuring stability and compatibility for development and performance-critical workloads.
- **Kernel Version**: Linux 6.8.0-49, offering the latest optimizations.
- **Windowing System**: X11, delivering graphical support for the GNOME environment.

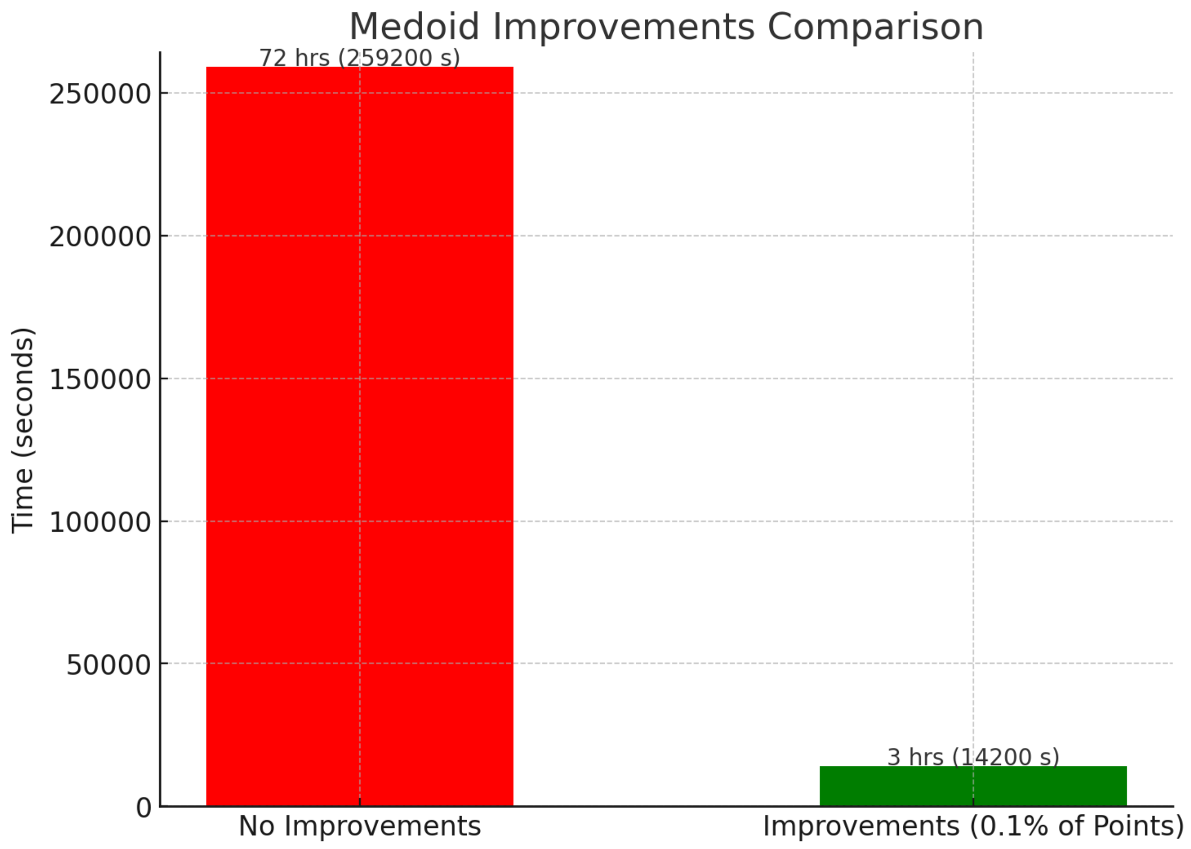
Medoid Optimization

Calculating True Medoid from All Points

Initially, we attempted to calculate the true medoid from all points in the dataset. However, this approach proved to be computationally expensive, especially for large datasets. The time complexity of this approach was O(n^2), where n is the number of points in the dataset. As a result, this approach was not feasible for large datasets.

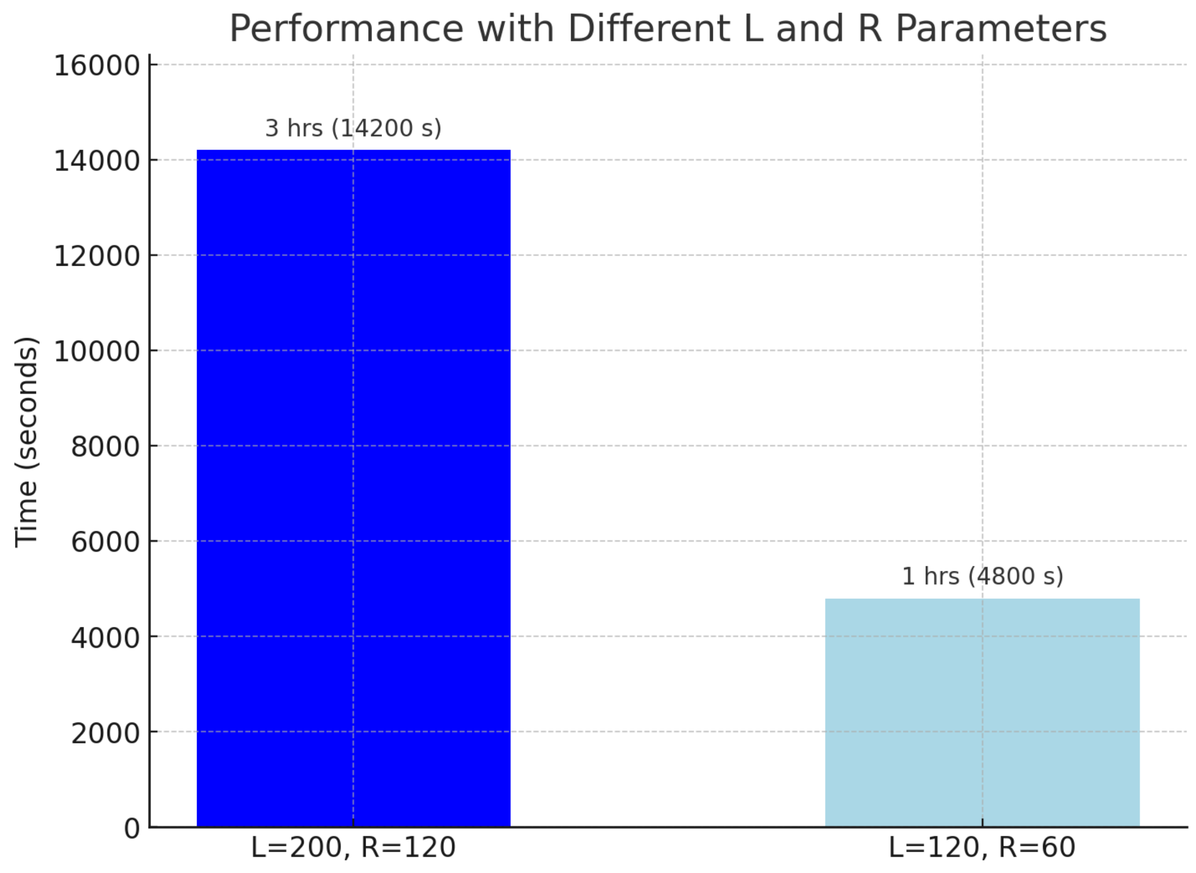
Sampled Calculation

Estimated time for calculating true medoid would be around three days.



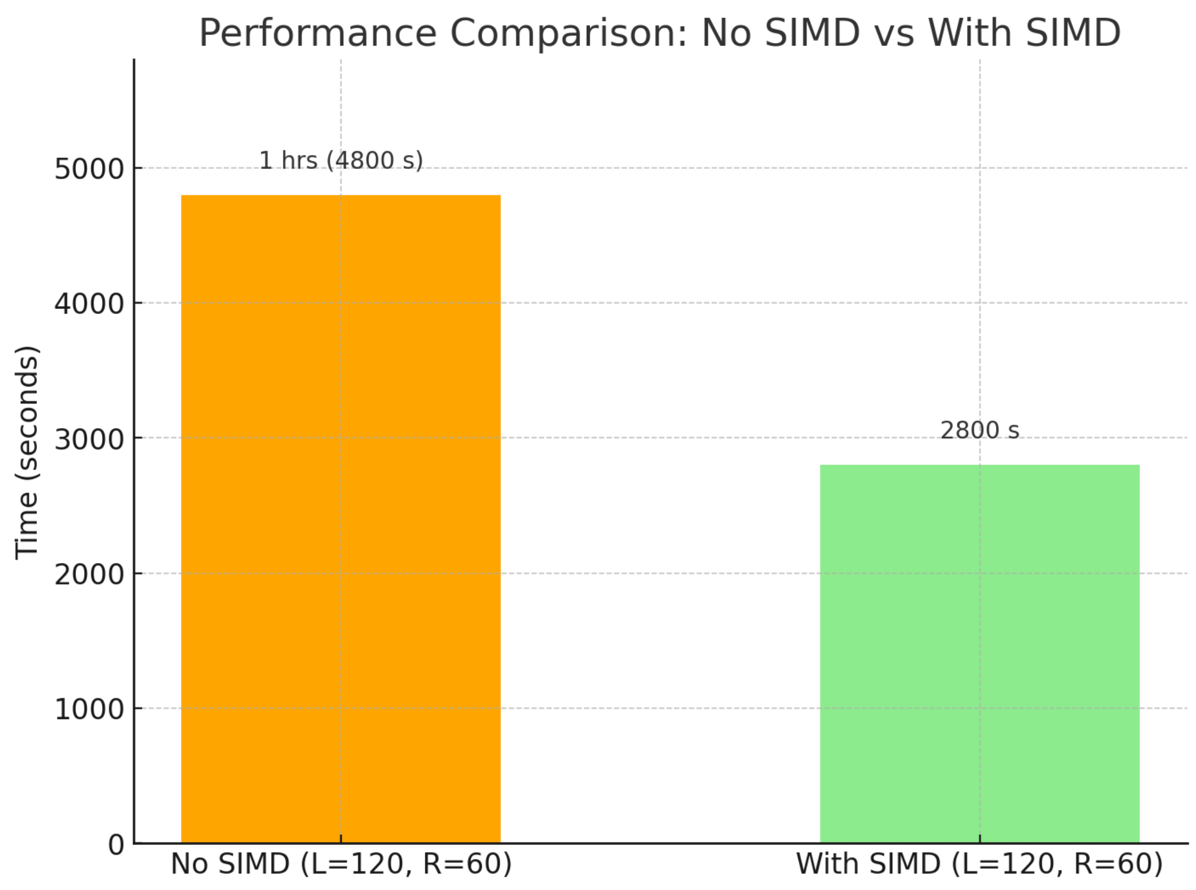
L and R different configuration optimization

The trade-off between precision and computation time revealed that using smaller values of L and R was more beneficial. The difference in precision between the two configurations was only **1%**, yet the reduction in computation time was substantial. This demonstrates that opting for lower L and R values can significantly improve efficiency without a noticeable loss in precision, making it a practical choice for performance-critical applications.



SIMD utilization for euclidean distance computations

The statistical analysis shows that a significant portion of the program's runtime is spent on distance computation. By utilizing SIMD, we observed a substantial performance improvement of approximately **41%**, highlighting its effectiveness in optimizing this critical part of the process.

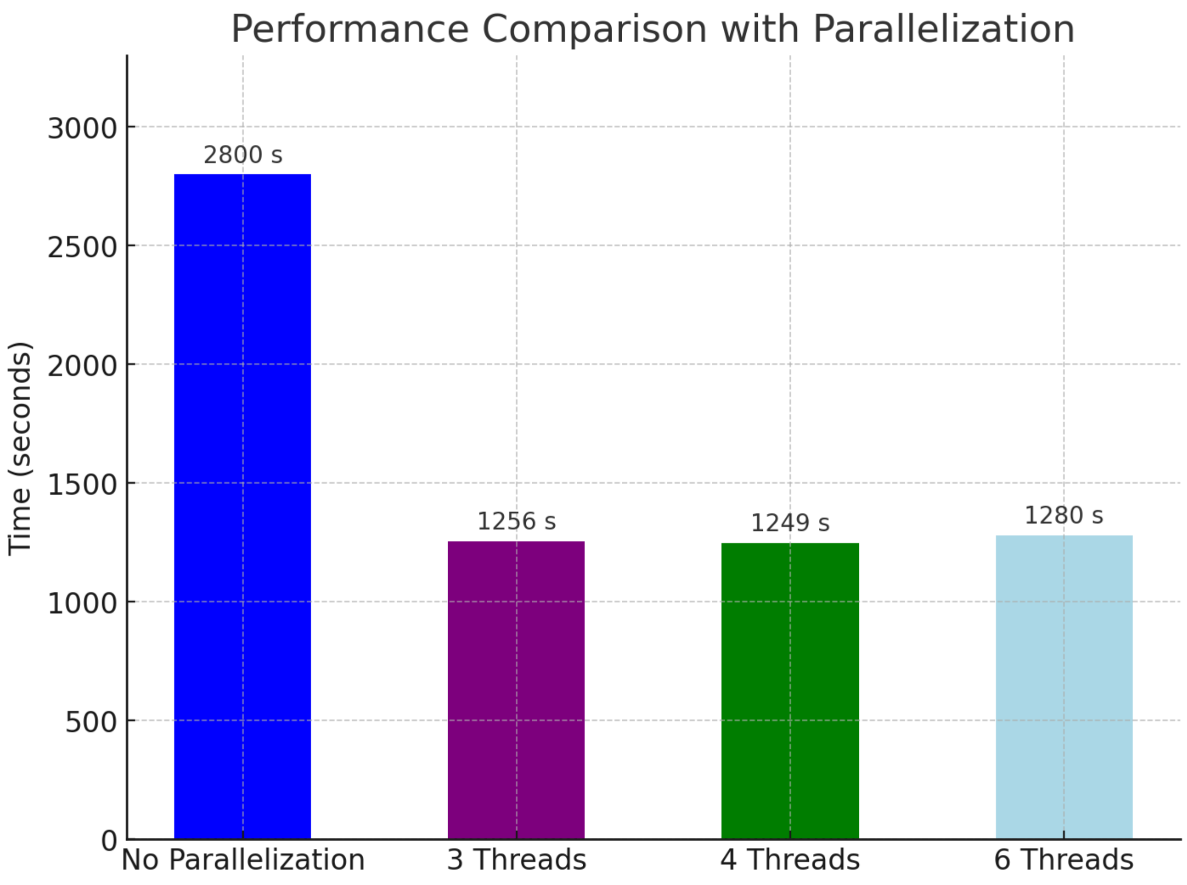


Parallelization of Filtered Vamana

The parallelization engine was implemented using threads, distributing the workload across different cores. With the assumption that points are associated with distinct filters, the parallelization was applied during the creation of the Vamana structure, processing each filter independently since they are uncorrelated.

To optimize performance, a descending sorting was performed based on the number of points in each filter, with the filter containing the most points assigned to the first thread, followed sequentially by the rest. This approach resulted in a significant performance improvement, particularly in this specific dataset, where approximately **40% of the points** belonged to a single filter. This led to the performance gain being directly proportional to the processing of the points in that dominant filter.

If the data were more evenly distributed across the filters, we would observe a greater performance gain from parallelism. However, due to the imbalance, the improvement plateaued after utilizing four threads. This also explains why no further performance gains were observed beyond four threads. The same phenomenon would be observed if the test were conducted with three threads.



Final recall for the filtered vamana

The recall calculation has been completed with the following results:

- **Total Points Found**: 339,259 out of 501,100.
- **Filtered Points Found**: 239,518 out of 249,500.
- **Overall Recall**: **67.70%**.
- **Filtered Recall**: **96.00%**.
- **Time Taken for Recall Calculation**: 7 seconds.
- **Avg time per query:** 0.0007 seconds

Stitched Vamana

Machine used for the tests:
CPU: AMD Ryzen 5 2600
RAM: 16GB

We will present the results of our experiments, which were conducted on two different file sizes: a contest file with 1M data points and a dummy file with 10K data points. We will also present the results for two different cases: (L=120, R=60) and (L=200, R=20). The cases were tested on 1 to 7 threads. We chose these cases to see how the recall and time of indexing change depending on the density of the graphs.

All the tests happened after the General Optimizations took place.

Medoid Optimization

Calculating True Medoid from All Points

Initially, we attempted to calculate the true medoid from all points in the dataset. However, this approach proved to be computationally expensive, especially for large datasets. The time complexity of this approach was O(n^2), where n is the number of points in the dataset. As a result, this approach was not feasible for large datasets.

Sampled Calculation

To reduce the computational complexity, we then tried sampled calculation of the medoid. We experimented with sampling 40%, 20%, 10%, and 1% of the points in the dataset. While this approach resulted in better times, it was still not optimal. Moreover, we did not notice any significant hits on the recall, which remained stable across different sampling rates.

Using a Random Point as Medoid

Finally, we tried using a random point as the medoid. This approach proved to be both fast and reliable. We did not notice any significant hits on the recall, which remained stable. This approach also had the advantage of being simple to implement and requiring minimal computational resources.

Conclusion

Our experiments with optimizations specific to Stitched Vamana suggest that using a random point as the medoid is a viable approach. This approach offers a good trade-off between computational complexity and recall, making it suitable for large datasets. We did not notice any significant hits on the recall, which remained stable across different approaches. These findings suggest that the choice of medoid calculation method has a significant impact on the performance of the Stitched Vamana algorithm, and that using a random point as the medoid is a reasonable choice.

Parallelization

To further improve the performance of the Stitched Vamana algorithm, we explored the use of parallelization. By dividing the computation across multiple threads, we can significantly speed up the processing time.

Parallelization of Graph Construction

In our implementation, we parallelized the construction of the graph by dividing the computation across multiple threads. Each thread is responsible for constructing a portion of the graph, and the results are combined at the end to produce the final graph. This approach allows us to take advantage of multi-core processors and reduce the overall processing time.

Thread Management

Our implementation uses a thread management system to coordinate the execution of multiple threads. Each thread is responsible for processing a subset of the data, and the results are combined at the end to produce the final output. By using a thread pool, we can efficiently manage the creation and destruction of threads, reducing the overhead of thread creation and improving overall performance.

Parallelization of Stitched Vamana

We also parallelized the Stitched Vamana algorithm itself, dividing the computation across multiple threads. Each thread processes a different portion of the data, and the results are combined at the end to produce the final output. This approach allows us to take advantage of multi-core processors and significantly speed up the processing time.

Distributing Filters to Threads

In our implementation, we took a dynamic approach to distributing the number of filters to the number of threads. Instead of assigning a fixed number of filters to each thread, we used a load factor to determine the optimal distribution.

Load Factor

The load factor is a measure of the computational load associated with each filter, which depends on the number of points in the filter. Filters with more points require more computational resources, and therefore have a higher load factor.

Dynamic Distribution

Based on the load factor, we dynamically distribute the filters to the threads. The number of filters assigned to each thread is proportional to the thread's available computational resources and the load factor of the filters. This approach ensures that each thread is utilized efficiently and that the workload is balanced across all threads.

Benefits

By using a dynamic distribution approach, we can take advantage of the varying computational loads of different filters and threads. This leads to several benefits, including:

Improved utilization of computational resources
Better load balancing across threads
Increased overall performance and efficiency

Example

For example, if we have 10 filters with varying numbers of points, and 4 threads with different computational resources, our approach would dynamically distribute the filters to the threads based on the load factor. The thread with the most available resources might be assigned 3 filters with high load factors, while the thread with the least available resources might be assigned 1 filter with a low load factor. This ensures that each thread is utilized efficiently and that the workload is balanced across all threads

Performance Improvements

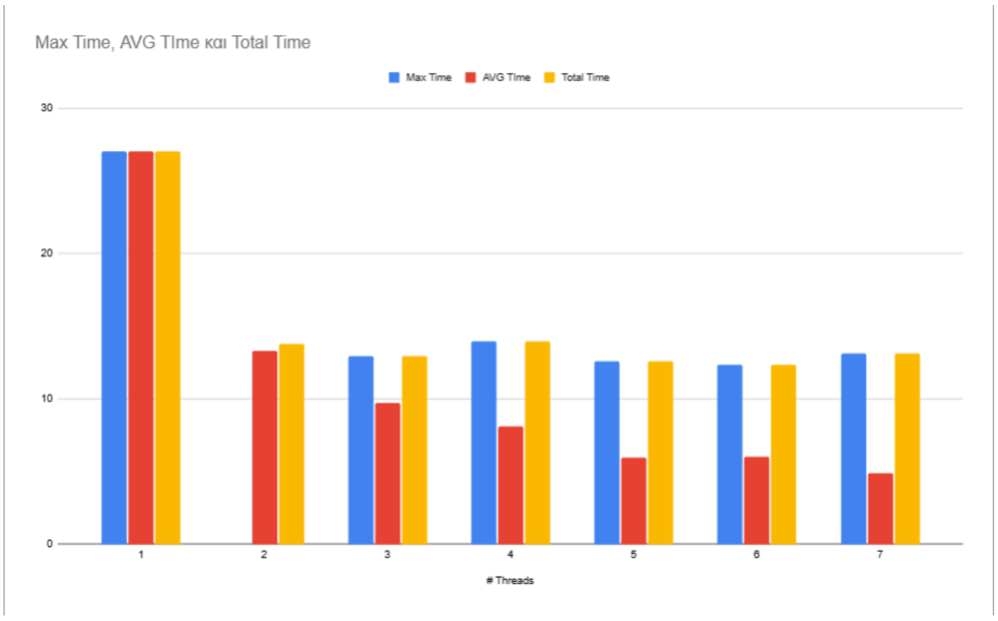
In this section, we will discuss the performance improvements we achieved through parallelization. We will present the results of our experiments, which were conducted on two

different file sizes: a contest file with 1M data points and a dummy file with 10K data points.
We will also present the results for two different cases: (L=120, R=60) and (L=200, R=20).
All these cases were tested on 1 to 7 threads.
We chose these cases to see how the recall and time of indexing change depending on the density of the graphs.

For the dummy file with 10K data points:

Case 1 (L = 120, R = 60)

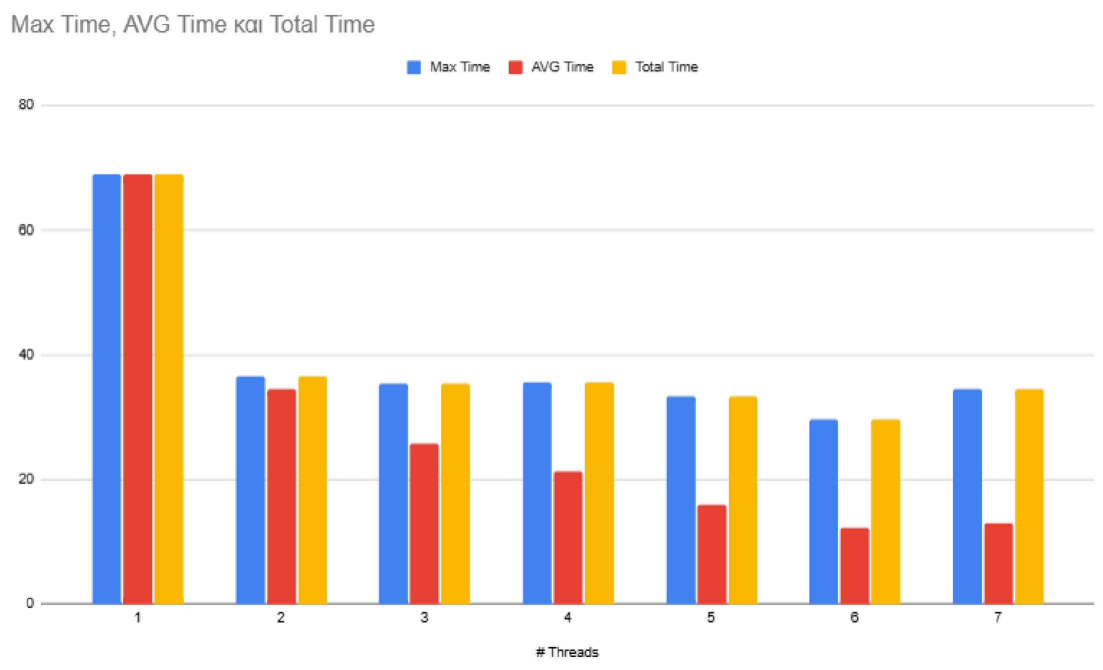
We noticed an immediate improvement moving from 1 to 2 threads with the time going from 27,05 seconds to 13,78 seconds. Thereafter, we saw slighter improvements to the time taken to index the graph by increasing the number of threads.



[Dataset 10K, R=60, L=120, Max Time of a thread, AVG time of threads, Total time of indexing]

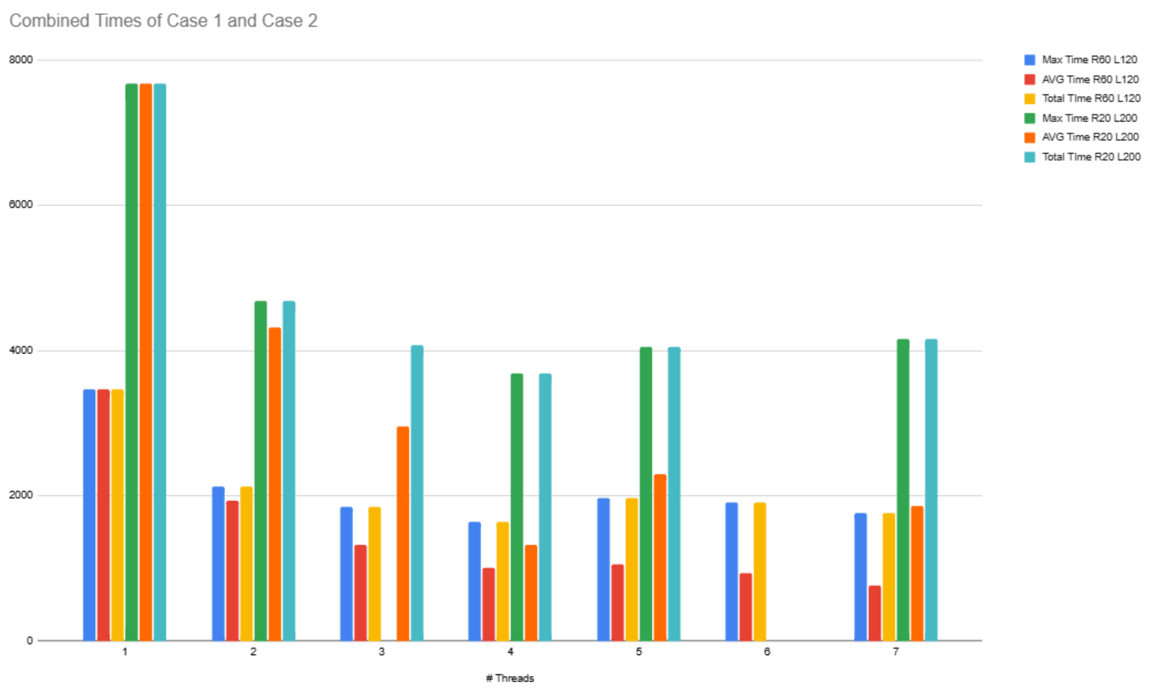
Case 2 (L = 200, R = 20)

Here, again we noticed an immediate improvement moving from 1 to 2 threads with the time going from 68,93 seconds to 36,57 seconds. Thereafter, we saw slighter improvements to time took to index the graph by increasing the number of threads.



[Dataset 10K, R=20, L=200, Max Time of a thread, AVG time of threads, Total time of indexing]

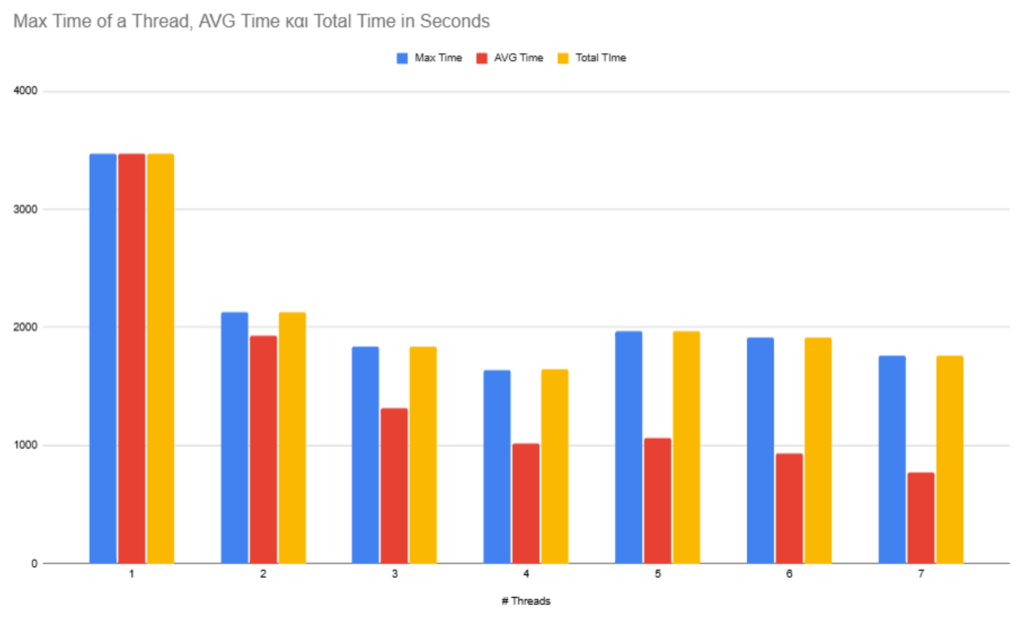
Below you can see the comparison better:



For the Competition file with 1M data points:

Case 1 (L = 120, R = 60)

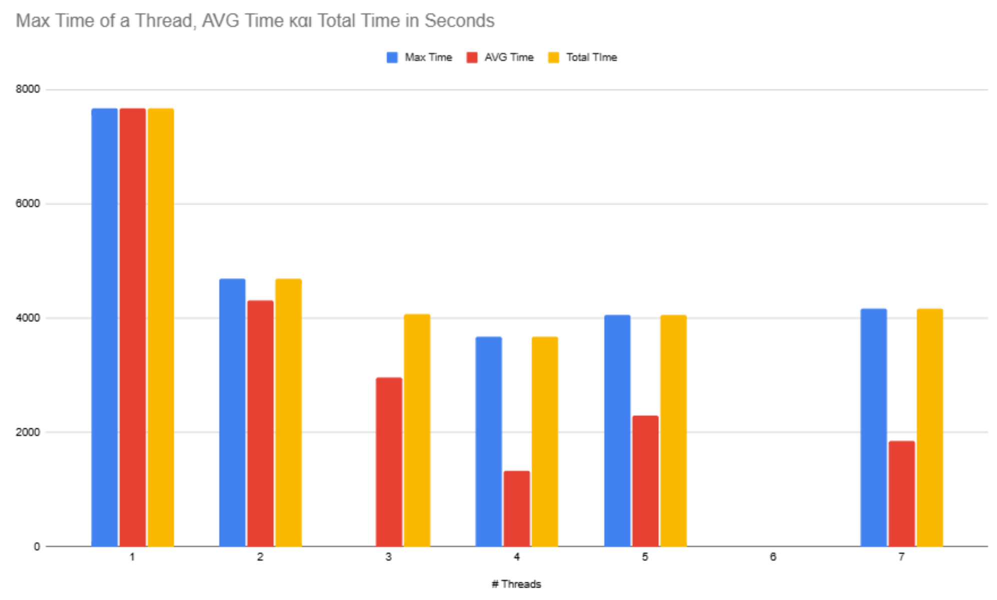
We noticed an immediate improvement moving from 1 to 2 threads with the time going from 3471,44 seconds or 57 minutes to 2129,59 seconds or 35,49 minutes. Thereafter, we continued to see improvements up until the 4 threads with index average of 1642,38 seconds or 27,3 minutes. After that the time increased until 6 threads to 1912,8 seconds and decreased again to 1757,7 seconds on 7 threads.



[Dataset 1M, R=60, L=120, Max Time of a thread, AVG time of threads, Total time of indexing]

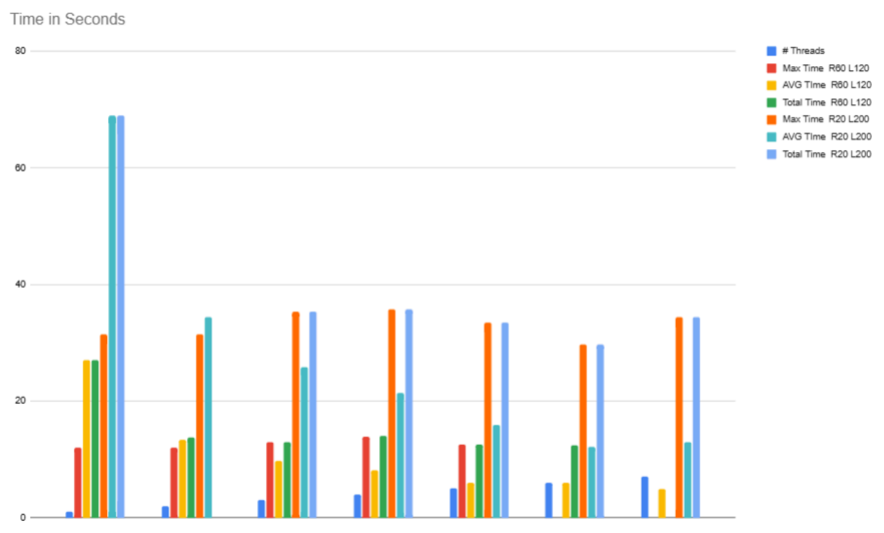
Case 2 (L = 200, R = 20)

Similarly to case 1, there was improvement until we reached 4 threads, with times starting at 7681,56 seconds and decreasing to 3680,89 seconds. Then the indexing time started increasing slightly until reaching 4164,58 seconds at 7 threads.



[Dataset 1M, R=20, L=2000, Max Time of a thread, AVG time of threads, Total time of indexing]

Below you can see the comparison better:



Recall

In this section, we will discuss the recall performance of our Stitched Vamana algorithm. We will present the results of our experiments, which were conducted on two different cases: (L=200, R=20) and (L=120, R=60) for small amounts of data(10K) and for larger amounts(1M).

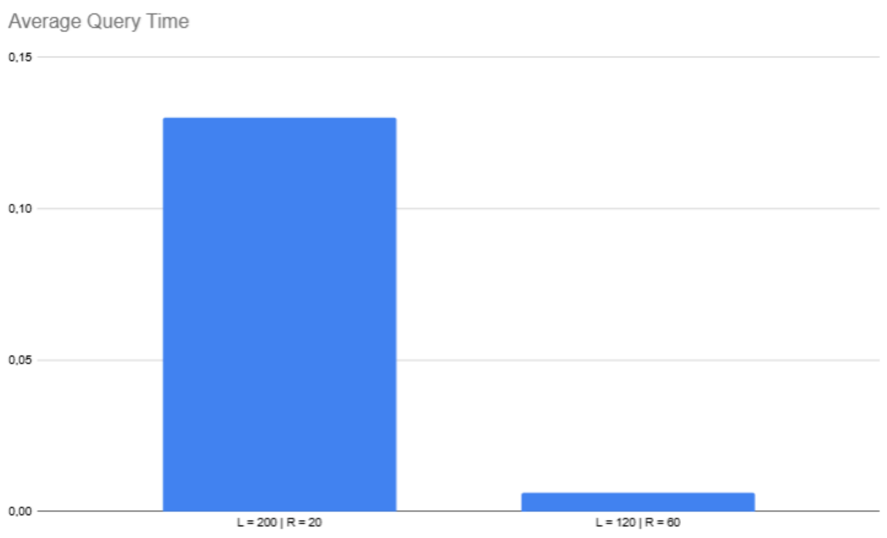
Small Data (10.000 Points):

Case	AVG time / query	Max Time / query	Total Recall Time	% Recall Accuracy	Number of Neighbors
L = 200 | R = 20	0,13	0,08	138,07	97,26	456.849
L = 120 | R = 60	0,006	0,04	65,34	98,11	460.869

Parameter Choices:

L=120 R=60: This setup achieves the highest recall (98.11%) with fast query times (average **~0.006-0.007 seconds** per query). It is the preferred choice for applications prioritizing high accuracy and responsiveness.

L=200 R=20: This configuration slightly reduces recall (97.26%) and increases indexing time due to higher computational demands. The overall performance makes it less ideal for small-scale datasets due to the significant increase of average time per query at **~0,13 seconds**.



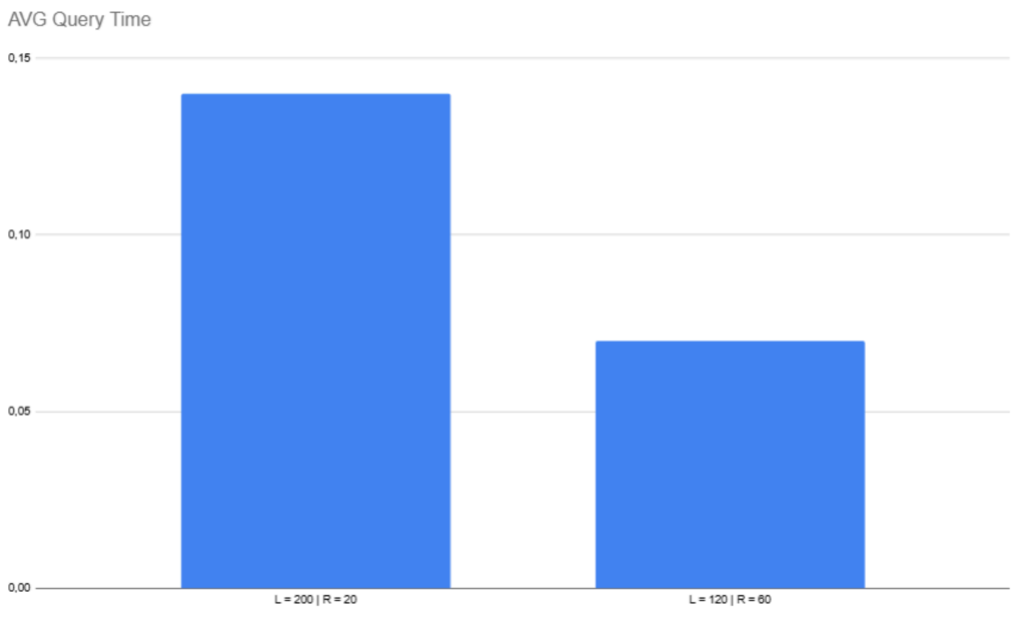
Large Data (1M) :

Case	AVG time / query	Max Time / query	Total Recall Time	% Recall Accuracy	Number of Neighbors
L = 200 | R = 20	0,14	0,78	1440,54	92,9	465.586
L = 120 | R = 60	0,07	0,49	742	96,53	483.758

Parameter Choices:

L=120 R=60: This setup consistently delivers higher recall (**~96.5%**) with shorter query times (average **~0.07 seconds** per query). It is well-suited for applications where accuracy and responsiveness are critical.

L=200 R=20: While this setup achieves comparable performance in smaller datasets, it introduces significant computational complexity for larger datasets. Indexing times are much longer, and recall drops to **~92.89%** with average time per query at **~0,14 seconds**.


AVG Query Time
0,15
0,10
0,05
0,00
L = 200 | R = 20
L = 120 | R = 60