Optimizing Multi-Attribute Filtering in Approximate Nearest Neighbor (ANN) Search

Zakaria CHOUKRI

December 27, 2024

Contents

1 Introduction

This project focuses on optimizing multi-attribute filtering in Approximate Nearest Neighbor (ANN) search using HNSW indexing. Each data point in the index can have at least 1000 labels (e.g., tags), and queries may specify one or two attributes as filters. The current filtering mechanism in HNSW gives the ability to evaluate custom functions for each visited point. Deciding if a point validates a query filter will require scanning of all point attributes and seeing if one or two of them validate the query tags (O(p) where p is the number of attributes per point). The project aims to improve the filtering process by:

1. Using bitsets to represent attributes of points for faster evaluation.
2. Further optimizing with Roaring Bitmaps, a compressed bitmap structure designed for high-performance set operations.

2 Created Files

I only created three c++ files for this project, which are:

- `example_Op.cpp`
- `example_bitset.cpp`
- `example_croaring.cpp`

Each one of these files does the filtering respectively, by using the O(p) approach that checks all attributes, the bitset approach, and its optimization with the Roaring Bitmaps library.

3 Benchmarking and Modularity

The code also contains timing mechanisms to benchmark each approach. It is also highly modular as I used nearly the same code that the developers of hnswlib used in their `example_filter.cpp` file, with the only difference being the filter class and function and how the attributes are represented.

4 Output on My Machine

4.1 1 Thousand Attributes

4.1.1 Bitset example run:

```
Search completed in 3.75686 seconds.
```

4.1.2 Roaring bitmap example run:

```
Search completed in 5.21877 seconds.
```

4.1.3 O(p) example run:

```
Search completed in 38.2249 seconds.
```

4.2 1 Million Attributes

4.2.1 Bitset example run:

```
Search completed in 310.597 seconds.
```

4.2.2 Roaring bitmap example run:

```
Search completed in 8.6197 seconds.
```

4.2.3 O(p) example run:

```
it took more than 90 minutes so I stopped it manually
```

5 Project Steps

I started the project by first reading about vector databases and indices from this link: `https://www.pinecone.io/learn/vector-database/`

Which led me to reading about ANN from here:

- `https://www.mongodb.com/resources/basics/ann-search`
- `https://mccormickml.com/2017/10/13/product-quantizer-tutorial-part-1/`

To finally understand this article about HNSW: `https://www.pinecone.io/learn/series/faiss/hnsw/`

6 O(p) Approach

The O(p) example is straight forward, just iterating through all the attributes in a linear fashion and stopping when there is a mismatch. I only did it in order to compare with it.

7 Bitset Approach

Bitsets are data structures that use bits to represent the presence or absence of an attribute, allowing for efficient storage and quick set operations. They are good because they enable constant-time membership tests. They also don't take as much memory as an array of booleans, since booleans each take 1 byte, whereas the elements of the bitset each take 1 bit. (1/8 memory)

I didn't have to do much research about bitsets because I had experience with them from competitive programming.

8 Roaring Bitmaps Approach

I then read about the Roaring Bitmap library. Roaring Bitmaps are improved versions of bitsets that use compression techniques to further reduce memory usage while maintaining high performance. They are better than bitsets because they are more efficient for sparse data which is the case with our project because usually only a small set of attributes is set and the rest is not .They also support fast set operations.

9 Difficulties

I encountered many difficulties and challenges in this project.
First, I had a hard time getting the hnswlib code to work on my machine. Even though I cloned the github repo, I had problems with CMake and building tools that had conflicting dependencies with what's on my computer. I overcame this by just taking the files that I will be needing instead of using the whole repository.
The second challenge was finding out what I needed to do exactly. I had to read articles about the subject and watch videos as well as ask ChatGPT for clarifications whenever I had a confusion with any concept.
Finally, after undestanding what needed to be done and fixing the errors I had to code. I had previously asked Anas a little bit about this part. And concluded that the implementation I would be doing would take place in one of the example files. So I created example files for each of the methods. I started by the O(p) method where I represented the attributes as a vector of booleans, and used for loops to do the check needed in the filter. The other methods had more efficient ways to check. After making sure that everything was working, I added the timing functionality in order to compare the different approaches.

10 Interpretation

In the 1,000 attributes case, the linear filter takes approximately 7 to 11 times more than the other two methods, with the fastest one being the bitset. This may seem weird, but it isn't because the size of the data is too small for the advantages of the Roaring Bitmap to appear.

With the 1,000,000 attributes case, the linear filter took more than 90 minutes, which caused me to stop it manually because it was not clear whether it was going to finish at all. The bitset performed well in just over 5 minutes, but the star of the show was the Roaring Bitmap that performed in only 8 seconds, making it extremely efficient.

11 Comparisons

Below are figures comparing the performance of the different approaches:

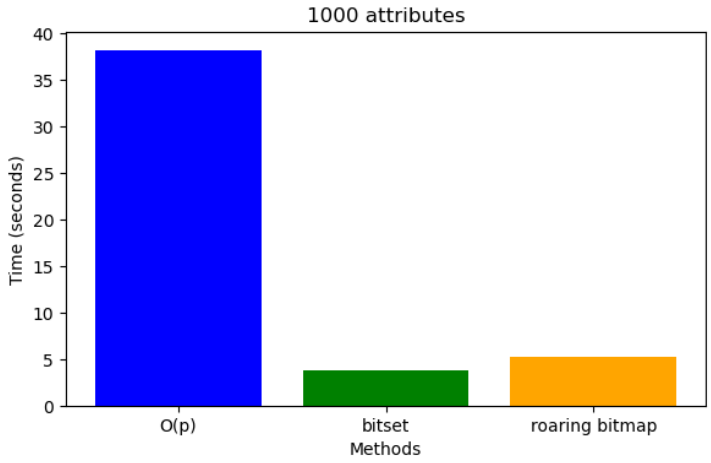

Figure 1: Performance with 1,000 attributes.

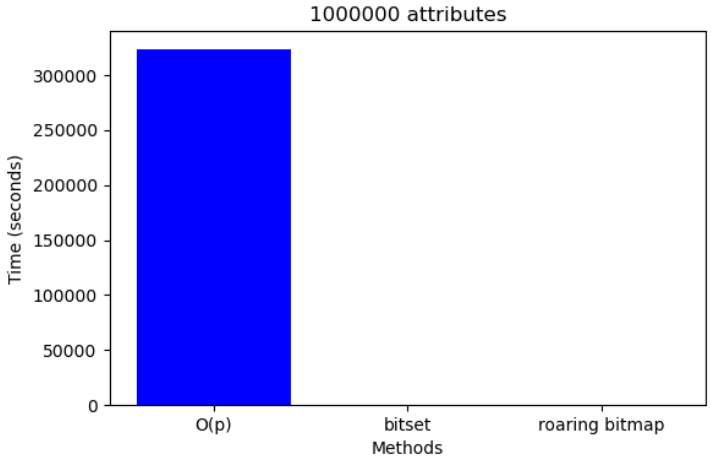

Figure 2: Performance with 1,000,000 attributes.

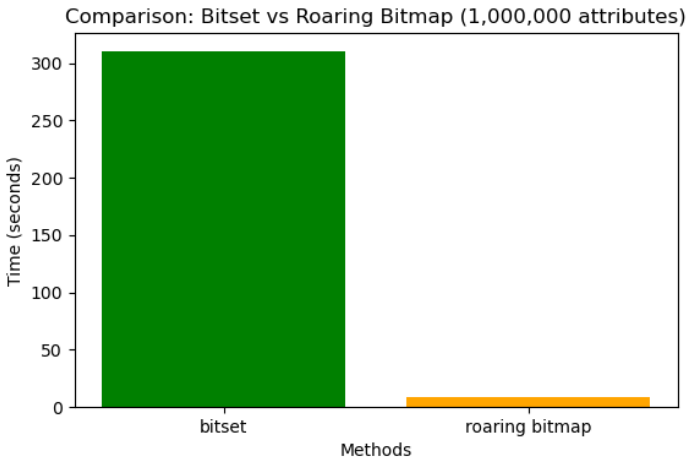

Figure 3: Performance with 1,000,000 attributes (only bitset and roaring bitmap).