Università di Genova
DIBRIS DIPARTIMENTO
DI INFORMATICA, BIOINGEGNERIA,
ROBOTICA E INGEGNERIA DEI SISTEMI

Distributed Computation of Approximate Nearest Neighbors

Christian Stingone

Master Thesis

Università di Genova, DIBRIS
Via Dodecaneso, 35
16146 Genova, Italy
https://www.dibris.unige.it/

MSc Computer Science
Software Security and Engineering Curriculum

Distributed Computation of Approximate Nearest Neighbors

Christian Stingone

Advisors: Matteo Dell'Amico
Miriam Redi

Examiner: Daniele D'Agostino

December, 2023

Abstract

This work aims to be a guide for users and companies who need to perform efficient and precise nearest-neighbor searches on large multi-dimensional datasets. To do this we introduced a family of algorithms called Approximate Nearest Neighbors, which allow approximate searches by providing a trade-off between accuracy and search execution speed.

The work focuses on the study of the best-known and most used Free and Open Source libraries and, for each of them, we analyzed the main and most efficient indexes. Since these solutions are native on a single machine and their ability to scale as the dataset grows is limited by the amount of available RAM, distributed solutions were also presented, which allow horizontal scaling. Once the state of the art had been studied, code was written to run the libraries on a dataset provided by the Wikimedia Foundation, made up of 6.4 million images. Characteristics such as the documentation and maturity of the libraries with regard to use in production environments were therefore analyzed. Following these implementations, various tests were performed trying to cover the majority of use cases, comparing and describing the capabilities of each index.

Finally, we have written a guide capable of responding to most of the needs of users and companies for the choice of libraries, indexes and parameters to be used depending on the reference dataset, obtaining high accuracy and low execution times, in the order of a few milliseconds.

Table of Contents

Chapter 1

Introduction

We live in a world increasingly surrounded by technology and, through it, we can bring innovation to all people. To do this, of course, we are faced with increasingly large datasets and the need to have increasingly performing and scalable systems in order to be able to handle the workload required today and, possibly, in the future. This is the goal that *Wikimedia Foundation* [wika], a collaborative foundation in this thesis, has been setting for years, providing non-profit systems to the world. One of their best-known products is *Wikipedia* [wikb], "the free encyclopedia that anyone can edit". Thanks to this collaboration and this portal, a new need arose, which allowed us to write this thesis. Their use-case is: Wikipedia is a free online encyclopedia where any user can upload the articles and information they deem appropriate. While doing that they can also upload pictures. This poses a problem as some of them may upload images under an incompatible copyright license. To overcome this, Wikipedia provides a service that, taking the picture that the user wants to upload as input, provides, as output, visually similar images, with compatible copyright licenses, present in their dataset. As can be easily imagined, their dataset is enormous and, therefore, their system must be able to perform fast and precise similarity searches on a considerable amount of data. This need has led us to evaluate a type of search algorithm useful for solving this problem, the *Approximate Nearest Neighbors*.

This family of algorithms is necessary for the resolution of this and many other use cases as it allows very fast approximate searches maintaining very high precision. Research in this field is all in all young with various solutions proposed in the scientific literature and, some, even in the form of free and open source libraries. Unfortunately, despite the various papers, the actual implementations of the algorithms presented are not many and, those present, are not always well documented or mature enough to be brought into a production environment. This is also noticeable through *ann-benchmarks* [annc, anna], the most famous benchmarks website in this field, which does not provide enough details on the algorithms used and the data collected, it is limited to being a simple comparison of graphs,

without an effective guide. This is due to the fact that there are very few companies that need to manage services that operate on huge datasets and in a very short time as in our case, making this domain less explored by most users or companies. In fact, we will have to carry out searches in a dataset of approximately 6.4 million images [data, datb] while maintaining very low execution times (possibly as low as the average latency of network communications, estimated to be between 20 and 100 milliseconds) in order to support the traffic of a portal with approximately 14 million daily users. Most of the works we are going to analyze are carried out by large companies such as Spotify, Meta or Microsoft.

Through this thesis, we therefore want to create something that is not present nowadays, a document to guide all those users and organizations who find themselves having to carry out research on an extremely large amount of data in a very short time. To do this we will begin with chapter 2, analyzing the most common, famous and used libraries to perform approximate searches on a single machine. This will allow us to see extremely efficient algorithms, which share the same limit, the amount of RAM. This will be a big obstacle since it is the one that defines the quantity of data present in the dataset on which we can train our indices. As this amount decreases, the accuracy will also decrease. To overcome this problem we can only perform vertical or horizontal scaling. Here we will enter the chapter 3 where we will see libraries perform approximate searches in a distributed manner, thus mitigating the problem imposed by the RAM. To do this we will present distributed versions of libraries already known in chapter 2 as well as more complex systems, thus falling into the field of vector databases. Once we finish the study of literature, in chapter 4, we will implement these solutions by integrating the libraries within our codebase [the] to evaluate the documentation, the maturity of the libraries and the effective usability. This will be necessary as the libraries provided do not always implement what is defined in the state of the art. We will therefore also have to implement features not present in the original libraries and integrate them. It will be in chapter 5 where we will actually execute the code to compare all the solutions previously presented by creating indexes on various single and distributed machines over different portions of the datasets and with different training phase sizes. These tests, performed on the Wikimedia Foundation dataset, will allow us to replicate a production environment. Through this work we will therefore show how to obtain optimal results even with approximate libraries, achieving high precision with few milliseconds response times. Thanks to these results we will be able to direct the reader towards the best library for their use case, making comparisons and, finally, drawing some conclusions, for each of them. To do everything that was presented we will only consider *Free and Open-Source Software* (FOSS).

Chapter 2

Approximate Nearest Neighbors

The nearest neighbor search (NNS) problem is defined as follows: given a set of n points $P \in \{p_1, p_2, \ldots, p_n\}$ in a metric space X and a query point $q \in X$, find the closest point in X to q [MP69]. The low-dimensional case is well-solved [GE89] so the main issue is that of dealing with the "curse of dimensionality" [Cla94]. The algorithms known to solve this problem are of two types, ones with high cost in the preprocessing phase while others in the query phase. As a solution to this, a more general problem called the ϵ-approximate nearest neighbors search was proposed. It is defined in the following way: find a point $p \in P$ that is a ϵ-nearest neighbor of the query q, that is for all $p' \in P, d(p, q) \leq (1 + \epsilon)d(p', q)$ [IM98].

The literature provides us with several algorithms that succeed in this aim, including hash based methods [DIIM04, JKG08, WTF09, XWL+11, WZ19, WZS+18, ZDW14], tree-based methods [Ben75, LMGY05, WWJ+14, ML14], graph-based methods [HAYSZ11, DCL11, WWZ+12, MY18], and hybrid methods [WL12, Iwa16, CWL+18], although, given the size of our dataset, memory has become the biggest bottleneck, not allowing a real use of some of these solutions. What makes our case difficult is the amount of data, which does not allow it to be saved in RAM without adopting compression techniques.

We can also distinguish only two types of algorithms useful for our case, namely *inverted index-based* and *graph-based*. Inverted index-based algorithms divide the dataset into clusters and perform queries between the query vector and representatives of the clusters. This reduces the search space and, therefore, makes the algorithm faster. Graph-based algorithms make a hybrid use of memory (primary and secondary memory) to save in RAM the spread-out graph [FXWC18] and the original vectors on the disk. When a query comes, it traverses the graph, represented by compressed vectors, using a greedy approach and then reranks the candidates according to the distance of the original vectors.

Despite the high accuracy and efficiency of these algorithms, we are still in the approximate realm as these solutions are classified as Approximate Nearest Neighbors. Both types of

algorithms have configurable values to choose the necessary trade-off between execution speed and accuracy. Obviously, the tuning of these parameters will have to be done with precision and adapting them to the individual datasets, through appropriate tests.

We will go into the details of the algorithms, costs and main parameters later in this section, analyzing some of the most used solutions, often developed by some of the largest companies in the world.

Before exploring how these algorithms works in depth, it is necessary to introduce a widespread encoding typology for this kind of problem, the Vector Embedding. The idea is to be able to represent any object within a vector space. By doing so, it is possible to subsequently understand whether two elements are similar or close to each other by looking at the values associated with them. As already discussed, this can lead to the problem of the "curse of dimensionality"; it is therefore necessary to implement techniques to reduce the dimensionality of this vector space. This space is generally referred to as "latent" because we don't necessarily have any prior notion of what the axes are and we don't mind. What we care about is that objects that are similar end up being close to each other. Although it is necessary to apply a vector embedding model to our dataset before using Approximate Nearest Neighbors searches, which technique or library to use is up to us as we will not conflict with the ANN algorithm.

2.1 Annoy

Annoy (Approximate Nearest Neighbors Oh Yeah) [Annf] is an ANN library written by Erik Bernhardsson for Spotify [Annd, Anne]. It was created to find similar songs and provide playlists with a homogeneous listening experience.

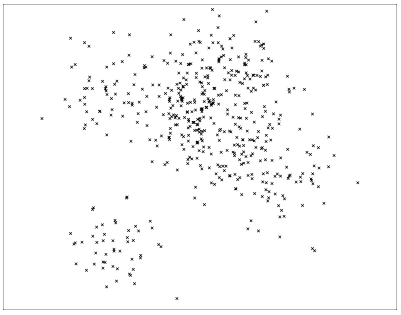
Following Erik Bernhardsson's presentation at the NYC ML meetup [Annb], we can divide the architecture of Annoy into two parts.

2.1.1 Tree Creation

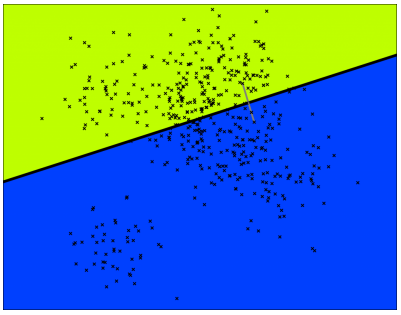
The first question is *why do we need a tree?* We are going to build a tree that lets us do queries in $O(log(n))$ time. In particular, we will use a data structure called *Random Projection Tree* [DF08]. We start by taking the dataset of Figure 2.1a to which we will have already applied the vector embedding model. Initially, we will apply on it a division by randomly extracting as many points as the dimensionality of the dataset, in this example we are working on a two-dimensional space, so we extract only two points. The points are extracted following a uniform probability distribution between the extremes of each dimension of the dataset. We use the hyperplane passing through the extracted points as a divisor. As shown in Figure 2.1b, the dataset has been divided into two partitions. This procedure will be repeated recursively in each newly created partition. As you can see in Figure 2.2a we have already obtained four divisions which can be represented as nodes/leaves of a binary tree Figure 2.2b. This procedure, performed repeatedly, will lead to obtaining a large number of partitions. Execution ends when each set contains at most K elements. We finally got a result shown in Figure 2.3. It is important to note that points that are close to each other have a good chance of being close in the resulting binary tree as well. In case this does not happen because, for example, two close points are separated by a hyperplane, this will reduce the precision of our approximate method, furthermore, it is sufficient to perform this procedure of creating the tree several times and compare the results to probably get a cluster very close to the correct one. This is due to the non-determinism in the choice of the hyperplane.

2.1.2 Search

At the end of the previous phase we obtained a tree, as shown in Figure 2.3. Analyzing this data structure we can see how each intermediate node represents a hyperplane, while each leaf represents a partition. Since this tree structure is binary, it is possible to search into it with a logarithmic cost, based on the height of the tree. The search procedure is as follows: we start from the root and ask ourselves how the query point relates to the

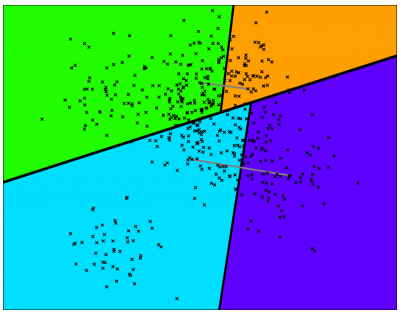


(a) Dataset represented in 2D (may have more dimensions).

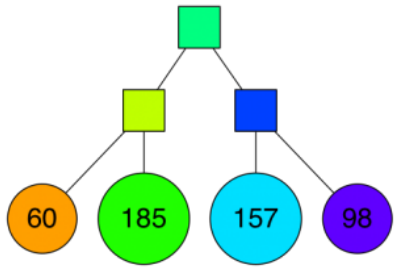


(b) First split due to a hyperplane, plotted as a black line.

Figure 2.1: Example dataset in 2-D space and first division. Image from [Annb].

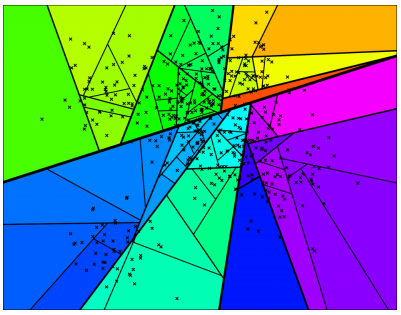


(a) Second split, another hyperplane for each partition.

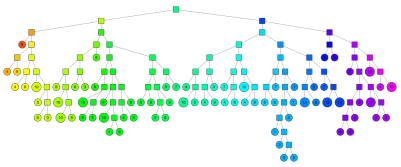


(b) Binary tree obtained from Figure 2.2a.

Figure 2.2: Tree construction. Image from [Annb].



(a) Dataset subdivided so that each set contains $K = 10$ elements



(b) Binary tree obtained from Figure 2.3a

Figure 2.3: Result of tree creation phase. Images from [Annb].

hyperplane that corresponds to the node we are referring to. We go down the appropriate branch and, once again finding ourselves confronted with an intermediate node, we repeat the work just performed. We will stop when we reach a leaf node, which contains a number of elements less than K.

Surely this procedure is very simple, but it leads to a question: what should we do if we wanted to return more elements than those present in the obtained partition? Among the possible solutions, the most popular are:

- Use a priority queue: this solution provides for having a threshold value that allows to choose whether to descend also along the branch that we would have previously discarded. This way we can search for multiple branches at the same time. This will be done by inserting the values into a priority queue and sorting them by the distance between the partition nodes and the query point. In this way we will continue the descent along the most useful branch, thus implementing a greedy algorithm. In this way we will have a larger set of results and we will therefore be able to obtain more than K
- Create a forest of trees: the idea behind this solution is to use non-determinism in the creation of the hyperplane by creating multiple trees, all different from each other. We will use a priority queue to keep in memory the trees we will search, sorted according to how far the query point is from the hyperplane corresponding to the node under consideration. In this way, for each tree, we will have only one descent path and we will always continue the search within the tree which will return the most accurate possible value. It is interesting to note how during the descent phase we never calculated the distance between points, but only the distance between the hyperplane and the query point. Once the search is completed we will obtain a much larger number of values and, finally, we will calculate the distance between them and the query point. In this way we will have a probabilistically better precision, as well as a larger number of elements on which to make the selection. Obviously the trade-off between precision and execution speed is given by the threshold that we are going to set. This threshold will tell us which trees to continue to consider in the descent and which ones, instead, to leave aside as not useful or of little significance.

As it is easy to understand, the second solution is the best as it not only solves the problem previously introduced but also offers greater precision as well as greater control over the trade-off between accuracy and execution speed. However, it is important to remember that, due to the "curse of dimensionality", it is advisable to decrease the dimensionality of the data before using Annoy as this algorithm suffers more from high dimensionality rather than from a large number of elements.

2.2 FAISS

FAISS is a library for efficient similarity searches and clustering of dense vectors. This library, developed by the Meta research team [Faia] was created to carry out searches on domains of several millions or billions of elements in very short times, even 10-100ms as reported in [MUS18]. The peculiarity of this solution is that if works with such large datasets that they would not be able to be loaded in RAM without adopting ad-hoc solutions. This algorithm, like all the other ANN algorithms, offers parameters with which it is possible to choose the compromise between execution speed and precision, as we will see later.

2.2.1 Product Quantization

Before working on the dataset we will need to compress it in some way, to do this we will use a technique called Product Quantization, first proposed by Gray [Gra84] and applied to ANN by Jégou et al. [JDS11]. This can be used to compress high-dimensional vectors, keeping the search operation simple with the compressed vectors and easily allowing the calculation of the distance between them and the original vector. This looks great, but how do we get a compressed vector? Through encoding.

To encode we need to define a procedure called Vector Quantization. In this method a function called quantizer q maps a D-dimensional vector $x \in R^D$ into a vector composed of values called centroids $q(x) \in C = \{c_i; i \in I\}$ where I is a finite set of values from 0 to $k-1$, c_i are the centroids and C is the codebook of size k. The set of vectors mapped to index i is called a Voronoi cell [CRSW22], so each vector belonging to a specific Voronoi cell will be encoded with the same centroid. In other words, a Voronoi cell of a centroid can be defined as the set of vectors whose nearest centroid is the one at the center of the cell. To calculate the quality of a centroid we have to use the mean squared error defined as $E_x[distance(q(x), x)^2]$ where the *distance* function is the Euclidean distance which compares the quantized vector with the original one. A near-optimal quantizer that can be used is the Lloyd quantizer, which corresponds to the k-means clustering algorithm and iteratively improves the accuracy of the centroid based on the elements it receives as input.

Vector Quantization must be applied to high-dimensional vectors, to do this each vector will be decomposed into m subvectors and an ad-hoc quantizer will be performed on each block $j \in \{1, \ldots, m\}$, this is called Product Quantization. To make it easier to understand this technique we can summarize it as follows:

$$\underbrace{x_1, \ldots, x_{D^*}}_{u_1(x)}, \ldots, \underbrace{x_{D-D^*+1}, \ldots, x_D}_{u_m(x)} \rightarrow q_1(u_1(x)), \ldots, q_m(u_m(x))$$

where q_j is the quantizer assigned to the j subvector. The encoded vector can be defined as the concatenation of centroids $C = C_1 \times \ldots \times C_m$. An interesting property of product quantization is that it is possible to approximate the input vector given the encoded vector, this feature is not used, however, by FAISS and therefore we will not go into detail, which is available in other sources [JDS11].

Regarding memory usage, Product Quantization is a method that, compared to others, allows to store data in memory even if the dataset grows. This is possible because the vectors resulting from the operations described above are not saved in memory, but in their place we will save $m \times k^*$, where k^* is the number of centroids for each subquantizer. In this way we will effectively write in memory $mk^*D^* = k^{1/m}D$ values, obtaining a remarkable efficiency.

2.2.2 Search

As far as the search is concerned, we must first find a way to calculate the distance between two encoded elements. To do this, the Asymmetric Distance Computation (ADC) is used.

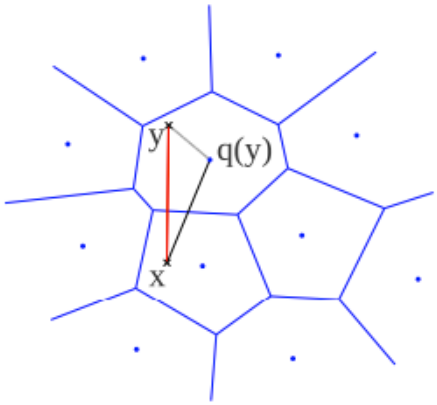


Figure 2.4: Asymmetric Distance Computation (ADC) between the query vector x and $q(y)$ that approximates $distance(x, y)$. Every cell is a Voronoi cell with a dot in the center that represents its centroid. Image from [JDS11].

The ADC calculates the distance between the query vector x and an already encoded vector $y = q(x)$. The $distance(x, y)$ can be calculated approximately as the distance $distance(x, q(y))$ Figure 2.4, defined as

$$distance(x, q(y)) = \sqrt{\sum_j distance(u_j(x), q_j(u_j(y)))^2} \tag{2.1}$$

Actually, in our case, we will not calculate the square root since this function is monotonically increasing and therefore applying it doesn't alter the ranking. To make equation Equation 2.1 efficient we have to divide the procedure into several steps, that correspond to the steps necessary to carry out a search. Initially (1) we are going to build the distance table, then (2) we take the distances we need for the calculation and (3) carry out the calculation of the distance.

As regards the costs of this research we have two sources of complexity. The first (1) concerns the construction of the distance table where the cost corresponds to $O(Dk)$, The second (2, 3) concerns the calculation of the distance between single centroids given two vectors with complexity $O(Nm)$. This results in a total complexity of $O(Dk+Nm)$, where D is the number of dimensions of the vectors, k is the size of each codeblock, N is the number of rows in the dataset, a value that can correspond to several million or billions, as well as being the only significant value and m is the number of subvectors in which each vector is decomposed.

2.2.3 More Advanced Index Solutions

A search as described so far would be linear concerning the number of elements of the dataset. This technique is not usually suitable for production use as it is inefficient since the ANN is used on datasets of several millions or billions of items. Instead, we will use some other techniques such as *Inverted Index*, *Inverted Multi-Index* and *Hierarchical Navigable Small World index*.

2.2.3.1 Inverted Index

To use the inverted index [JDS11, MUS18] we first need to take the full dataset and split it into J disjoint buckets $X_1, \ldots, X_J$ through k-means. At each bucket is assigned a representative vector $u_j \in R^D$ and the distance between it and each element of its bucket is calculated. The result of $x - u_j$, where x is a vector of the bucket, is encoded with product quantization and saved in the j^{th} bucket's posting list. In the search phase, given a vector $y \in R^D$, the distance between y and u_j is calculated. This allows us to choose the closest bucket and, therefore, to query its posting list. Subsequently, the nearest neighbors are searched using the ADC method between the residual vector $y - u_j$ and the vectors in the posting list. This algorithm is efficient as it reduces the search space since only the elements of the chosen bucket are considered, allowing us efficient searches even on datasets with 10^9 elements.

2.2.3.2 Inverted Multi-Index

To use this index type we first need to perform the Product Quantization as reported in subsection 2.2.1, for simplicity we give an example in which we will proceed by dividing each vector into two subvectors. We define $p_i = [p_i^1 \ p_i^2]$ the decomposition of a vector $p_i \in R^D$ of the dataset, where $p_i^1 \in R^{D/2}$ and $p_i^2 \in R^{D/2}$, remembering that D is the number of dimensions in the dataset. We call $U = \{u_1, \ldots, u_k\}$ the codeblock related to p_i^1 and $V = \{v1, \ldots, v_k\}$ the codeblock related to p_i^2. By performing product quantization we will create a $k \times k$ matrix containing all possible pairs of codewords $(u_i, v_j) \mid i \in \{1, \ldots, k\} \wedge j \in \{1, \ldots, k\}$. We will identify each element of the matrix as W_{ij}.

Once we understand how to create this data structure, we need to ask ourselves how to query it. Given a query vector $q = [q^1 \ q^2] \in R^D$ and a desired output list of size $T << N$, we will query the matrix to find the T elements closest to the vector q. First, we're going to find the codebook that q^1 and q^2 belong to. To do this it is possible to use an exhaustive search since the number of elements of U and V is typically small. We therefore denote with $\alpha(k)$ the index of the element closest to q^1 and with $\beta(k)$ the index of the element closest to q^2, while we are going to define two functions to calculate the distance as $r(k) = distance_1(q^1, u_{\alpha(k)})$ and $s(k) = distance_2(q^2, v_{\beta(k)})$. We will now calculate the distances $distance(q, [u_{\alpha(i)} \ v_{\beta(j)}])$ for each i and j , indices of the matrix . The value obtained will be saved in cell W_{ij} as can be seen in the top part of Figure 2.5. Finally, we will visit the data structure obtained greedily by selecting each time the elements with a smaller distance from q using the multi-sequence algorithm as reported in [BL15] and shown in the bottom of Figure 2.5. This will go on until the output list of length T will be filled with elements.

Let's now analyze the costs of this solution. k^2 lists will be kept in memory, many of which are empty since they contain no elements. We can refer to this phenomenon as a non-uniform distribution of list lengths. This trade-off is, in any case, necessary as we are able to have greater precision given by the sampling density. This high density still remains a problem if we decide to increase the number of multi-indexes used, thus increasing the number of dimensions into which we will divide the vector, as well as increasing the number of empty lists. This leads us to conclude that, although a smaller number of dimensions leads to longer quantization times, dividing the vectors into two groups of dimensions could already be a good compromise between precision, speed and memory usage. From a temporal point of view, this algorithm used many vector instructions which on modern CPUs can execute really fast in comparison with scalar ones.

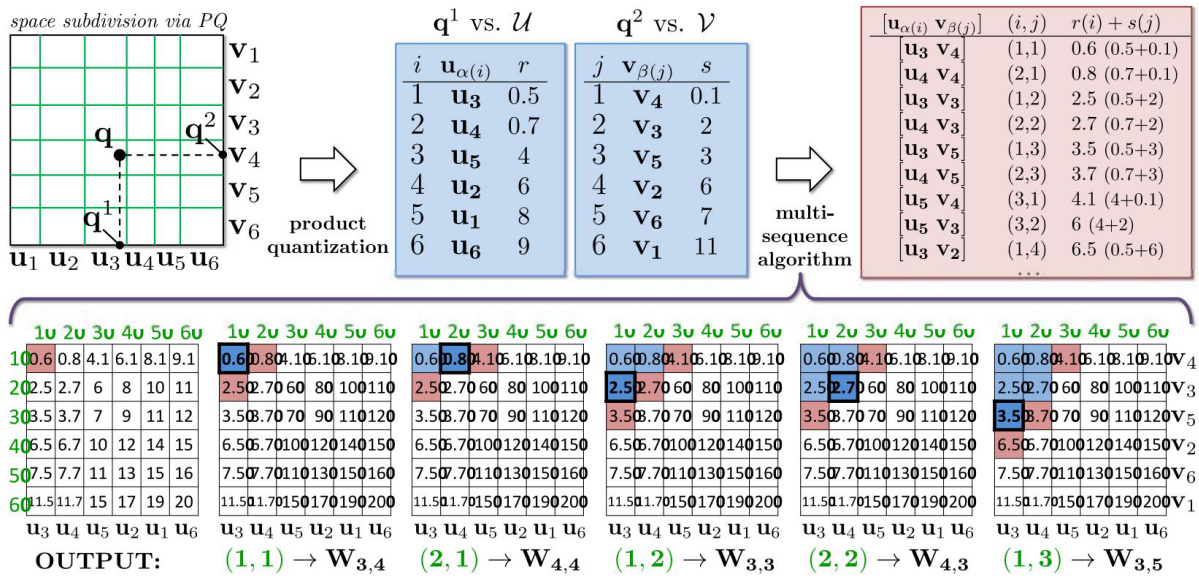


Figure 2.5: **Top part:** Find the codebook for q^1 and q^2, compute the distances and insert them into the matrix. **Bottom part:** Multi-sequence algorithm on the matrix to retrieve the centroids with a smaller distance from the query point. Image from [BL15].

2.2.3.3 Hierarchical Navigable Small World Index

The HNSW index (Hierarchical Navigable Small World) [MY18] is a type of index called *graph index* since it is based on a graph structure. To understand how it works, it is first necessary to introduce the concept on which it is based, i.e. Navigable Small Worlds, introduced for the first time by J. Kleinberg [Kle00b, Kle00a]. This type of graph is being studied for real-life network analysis, thus trying to represent networks with logarithmic or polylogarithmic scaling of the greedy graph routing. What is interesting about Kleinberg's NSW is that a node can have two types of arcs: *lattice edges*, i.e. short-distance arcs, and *long ranges*, i.e. more improbable arcs that connect very distant nodes. The probability with which two distant nodes u, v are connected corresponds to $P(u \to v) = \frac{1}{distance(u,v)^r}$, we can therefore note that r is the constant that represents the particularity of these types of graphs. Through this representation we can see how a graph with many nodes, even billions, can be traversed with a few steps, defined as *degrees of separation*.

The idea of a navigable small world has inspired an ANN approach [MPLK14] in which the NSW graph is built by connecting nodes to others in their neighborhood, and a node's neighbors are searched by navigating the graph, following first edges that lead closer to the query node. To overcome the risk of falling in local minima with a greedy approach, search is conducted on a parametric number (in general, between 50 and 200) of promising nodes. This approach, however, has the problem of missing the long-range "lattice" edges

discussed by Kleinberg, and for large datasets the search may take a very long time because a large number of short-range links need to be followed.

This is where a similar data structure, which manages to overcome this problem, the Hierarchical Navigable Small World (HNSW) [MY18], comes into play. The basic idea is not to have a single graph on which to perform all searches, but to have a hierarchical structure of graphs, called *layers*. The "bottom" layer (layer 0) corresponds to the NSW seen before and contains all nodes; each layer $i+1$ contains a sample of the nodes of layer i. As a result, each layer will connect nodes at different distances. The "upper" layer, i.e. the one furthest from layer 0, will contain the longest links. The search algorithm will work like the one for NSW until it finds the local minimum of the graph present in the layer. Once this is done, it descends hierarchically to the underlying layer, resuming the search from the previously selected node as shown in Figure 2.6. This procedure will continue until layer 0 is reached, which contains the entire graph, but at this point the nodes to check will be few and the query result will be easy to find. The degree for each layer can be made constant, increasing the number of layers if necessary, and the cost of this search is in practice polylogarithmic in real-world use cases. The cost of the memory used corresponds to the totality of the connections between the nodes, similarly to NSW in layer 0, giving us $O(Nk)$ where N is the number of nodes and k is the number of edges.

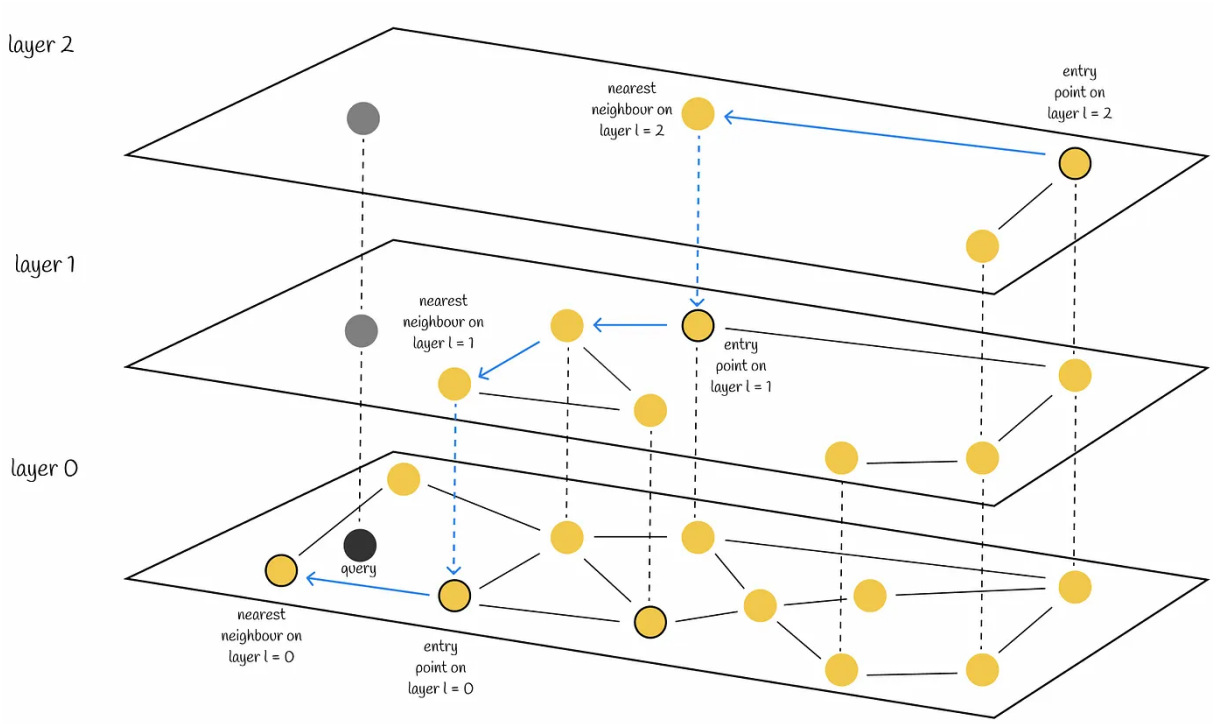


Figure 2.6: HNSW search example. Image from [hns].

2.3 SPTAG

SPTAG (Space Partition Tree And Graph) is a library [CWL+18] for large-scale vector approximate nearest neighbor search, created by Microsoft as an alternative version to their previous library, called DiskANN [Dis]. Unlike other technologies, this solution involves the use of both RAM memory and the physical disk, efficiently managing which and how much information to save on them. SPTAG offers three solutions: KD-Tree, Balanced K-means Tree and SPANN. We will only analyze the last two as KDT has been shown to be inefficient for large datasets and, therefore, for our use case, as reported in the Introduction of the library Github page [CWL+18].

2.3.1 Balanced K-means Tree

The *Balanced K-means Tree* term was not clearly defined in the documentation, and we could only find information about this index by looking through the papers created by the SPTAG authors. We suppose the descriptions we use match the behavior of the library, but we do not consider it as part of this thesis' scope to reverse engineer the SPTAG code to verify whether the implementation matches the descriptions provided. This solution consists of two parts, the Balanced K-means Tree [NS06] and the Relative Neighborhood Graph [WL12].

The first consists of creating a K-means tree and keeping it balanced in height and number of elements within the leaves. To do this, in the portion of the dataset on which the training will be carried, the K-means algorithm will be executed allowing us to obtain K clusters, each identified by a vector. In each cluster obtained from the execution of the algorithm, the K-means will be performed again to divide it into more clusters. This procedure will be repeated recursively, forming a tree where each node is identified by the cluster identification vector while the leaves correspond to the final clusters, which will effectively contain secondary memory pointing to all the vectors of the dataset saved adjacently in lists.

The second phase, i.e. the one relating to the Relative Neighborhood Graph, consists of analyzing more clusters and elements than those returned from a simple search in the tree obtained in the first phase. When we reach the cluster corresponding to the leaf of the Balanced K-means Tree we create a queue where we insert all the most promising elements to visit in an orderly manner. In addition to inserting the elements of the same cluster to which they belong, we also insert the elements of neighboring clusters. In this way it is possible to expand the search and obtain precise results even with values that live on the cluster boundary. In addition to this optimization, a caching technique has also been studied where, given a query point, we save a previous descent within the tree. This allows us to reduce the number of checks and operations necessary to reach the reference cluster

and, once reached, continue the exploration of the graph via the queue.

2.3.2 SPANN

This solution [CZW+21] mainly wants to solve three problems common to many approximate nearest neighbor search algorithms:

- Posting length limitation: when a dataset is divided into sets (or clusters), we will often have a very high variance in the number of elements within them. This makes the data structure, whatever it is, unbalanced. Limiting the size of these sets, balancing their cardinality, is a necessary solution to ensure efficiency.
- Boundary issue: once the dataset has been analyzed and divided into the data structure, we may notice how some elements have been assigned to one group rather than another since a check has been made on the distance between them and the groups' representative elements (often obtained via k-means). This is certainly correct, but, with query points that are positioned on multiple set boundaries, we may not get the best result because we may search only into some sets, maybe the ones that do not contain the value described above, as can be seen in Figure 2.7. To overcome the Boundary issue we will assign an element to several groups only if the distance from it to them is similar among all.
- Diverse search difficulty: this problem identifies a difficulty in executing some queries rather than others since for each query we will have a different number of sets to check. The goal is to reduce the number of accesses to the data structure avoiding checking some sets that will not help in reaching a good result.

2.3.2.1 Index Structure

The methodology used for the data structure follows that of the inverted index with the difference that no lossy data compression techniques will be adopted as there will be no restrictions on memory, a hybrid solution between RAM and secondary memory is used. The index structure foresees the division of vectors into N posting lists. Once the centroids relating to the posting lists have been calculated, using k-means, the vectors closest to the centroids are extracted and saved in memory with a direct reference to the lists present in the secondary memory. The use of this data structure allows efficient access to the relevant list, without wasting accesses in secondary memory (this operation on this type of memory is expensive). It is easy to understand that the longer a list is, the more accesses we will have to make to find the elements of our interest. This problem was previously introduced

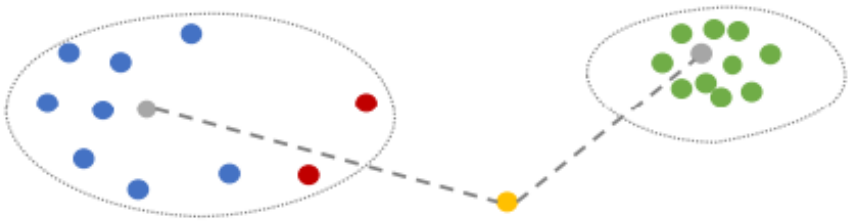


Figure 2.7: The image depicts a search, without the feature developed to overcome the Boundary issue, where the query point is colored in yellow. Comparing this point to the representative values of both groups we will notice how the green group will be closer. This leads us to scan the green cluster, although there are two elements of the blue group that are closer to the query point, highlighted in red. These elements, considered boundary elements, could be part of both clusters, thus allowing a more precise and efficient search. Image from [CZW+21].

in subsection 2.3.2 and solved through the Posting length limitation, which will use a multi-constraint balanced clustering algorithm [20119] to balance the lists. The idea is to minimize the variance on the number of elements within the lists through $\sum_{i=1}^{N}(|X_i| - \frac{|X|}{N})^2$ where X is the set of data vectors, N is the number of posting lists and X_i is the posting list with index i. This brings the advantage of not having to scroll through extremely long lists, but this is not the only problem since, despite a fixed list length, it may be necessary to access more lists to obtain a satisfactory result. This problem, previously presented under the name of Boundary issue in subsection 2.3.2, is partially solved by the Posting list expansion technique. This method uses the multi-cluster assignment solution for boundary vectors which is summarized by the formula Equation 2.2

$$x \in X_{ij} \iff Distance(x, c_{ij}) \leq (1 + \epsilon_1) \times Distance(x, c_{i1}) \tag{2.2}$$

where c are the representative vectors and c_{i1} is the closest one. This means that duplicate vectors will only be those on the boundary, while those close to the centroid will be reported only once. However, this solution risks not being efficient given the excessive duplication of vectors, to overcome this the RNG rule is followed [Tou80]. What this rule wants to do is to decrease the number of duplicate vectors in lists represented by very close centroids, in order to make a future search more efficient by expanding the number of different vectors that could be seen. To do this we will skip cluster ij for vector x if $Distance(c_{ij}, x) > Distance(c_{ij-1}, c_{ij})$.

2.3.2.2 Search

As far as the search phase is concerned, we have only one problem, previously described in subsection 2.3.2, to solve. To do this we need a technique called Query-aware dynamic pruning which consists of checking only the lists represented by a centroid close to the query point as much as the closest centroid, as reported in Equation 2.3, performing a distance check as in Equation 2.2.

$$q \xrightarrow{search} X_{ij} \iff Distance(q, c_{ij}) \leq (1 + \epsilon_2) \times Distance(q, c_{i1}) \tag{2.3}$$

where c are the representative vectors and c_{i1} is the closest one.

Chapter 3

Distributed Approximate Nearest Neighbors

In chapter 2 we have seen various libraries and techniques to perform an Approximate Nearest Neighbors search efficiently. These algorithms have also overcome some non-trivial problems related to the size of the dataset; due to the dataset enlargement, it is necessary to use more primary memory and disk space. However, this has limitations despite these spatial reduction characteristics, it will not be possible to operate on increasingly large datasets without reaching the physical limit (RAM) of the machine on which the algorithms are running. Once the limit imposed by the RAM has been exceeded, we will be forced to train the data structures on only a portion of the dataset. The remaining vectors will still be inserted, but they will not improve the state of the index, thus decreasing its precision. Furthermore, the previously found solutions are not full-fledged systems capable of operating at 360 degrees and capable of scaling as the dataset grows.

The question arises, *can't we do better?* To obtain an affirmative answer, we will have to leave single-machine solutions behind us to study systems capable of scaling and which do not suffer from some, if not all, of the previously mentioned problems, the distributed ones. We will need some complex systems such as those we are going to analyze in this section. Each of the next solutions will solve one or more of the problems mentioned above, thus providing a choice based on the most convenient trade-off in reference to one's use case.

3.1 Distributed FAISS

This library has already been covered in its single-machine version in section 2.2. Although its efficiency has already been demonstrated in the previous chapter, this may not be sufficient by the time we are going to approach, if not surpass, the physical limits of the machine. To work around this problem, a distributed version of this library [DFa] based on the work by Piktus et al. [PPK+21] has been made available. To get a general point of view on how it works, we can refer to Figure 3.1; we will go into in depth in each phase in the sections below.

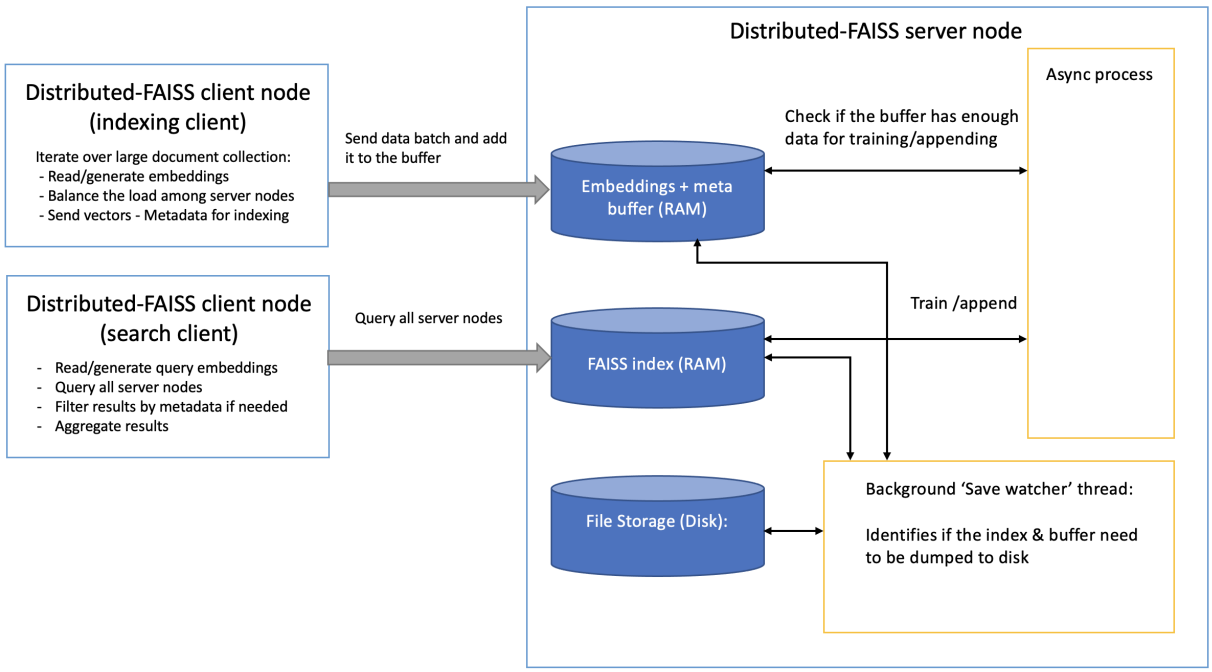


Figure 3.1: Behavior of the system. Image from [DFa].

3.1.1 Index Creation

We can logically divide the system into two main parts: *client* and *server*. The clients will take input blocks of vectors; these will be, through load-balancing policies, forwarded to the servers through an RPC library which will insert them inside the indexes as can be seen in Figure 3.2. The underlying indexes use FAISS; we can therefore summarize this library as a distributed wrapper.

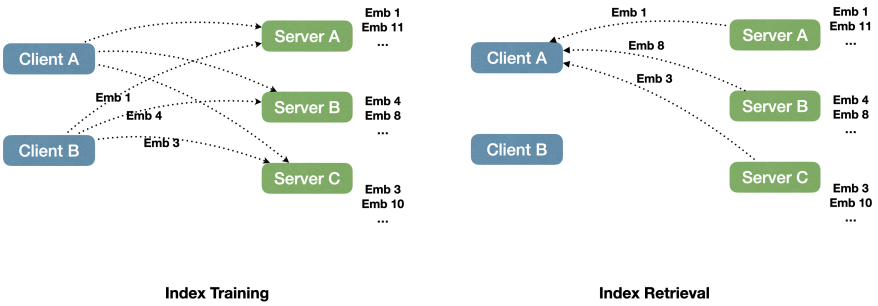


Figure 3.2: Index creation phase. Image from [DFa].

3.1.2 Search

As far as the search phase is concerned, this occurs similarly to MapReduce [DG08]. A client queries all servers for the necessary nearest neighbors. These will be merged and filtered on the client side to then get the best available neighbors.

3.2 Distributed SPTAG

This library has already been covered in its single-machine version in section 2.3. Although very efficient, the distributed version is quite similar to its single-machine version. What makes this library special, however, is that computation can be distributed on multiple machines in a very simple way.

3.2.1 Distributed Computation

Referring to subsubsection 2.3.2.1, once the index creation phase has been completed, we will have obtained an easily divisible structure. We will split the dataset into M posting lists where M is the number of machines and assign each posting list to a different machine. During the search phase we will only interrogate the machines on which the lists of our interests reside, parallelizing this procedure. This paradigm resembles MapReduce [DG08], an important framework for distributed computing. This type of solution, however, adds a problem to consider: some machines may become *hot-spots* as they are always queried meanwhile others may receive a much lighter workload.

3.2.2 Avoiding Hot-spots

To avoid the aforementioned hot spots we will change the approach partitioning the dataset into K partitions where $K \gg M$, remembering that M is the number of machines, and use the best-fit bin-packing algorithm [DS14] to pack K partitions into M machines. In this way we will be able to balance the queries on all machines, performing an efficient parallelization. This solution manages to scale very well allowing us to support much greater quantities of queries, even if we need to carefully evaluate the parameters K and M to obtain a fair trade-off between workload on each machine and speed of execution, considering the slowdowns due to the data aggregation phase.

Despite this solution to avoid hot-spots, we will notice in chapter 4 that this implementation is not present in the library, forcing us to create a custom version.

3.3 Milvus

Milvus is an open-source [Mila] vector database that focuses on providing an efficient system and architecture for similarity searches and AI-powered applications. The need that Milvus tries to satisfy is to provide a complete system capable of operating on huge dynamic datasets, providing solutions that can run in a distributed way. Milvus is based on the FAISS library (section 2.2), to which a few modifications have been applied.

3.3.1 System Design

Milvus consists of a structure divided into 3 main components: Query engine, GPU engine and Storage engine as shown in Figure 3.3.

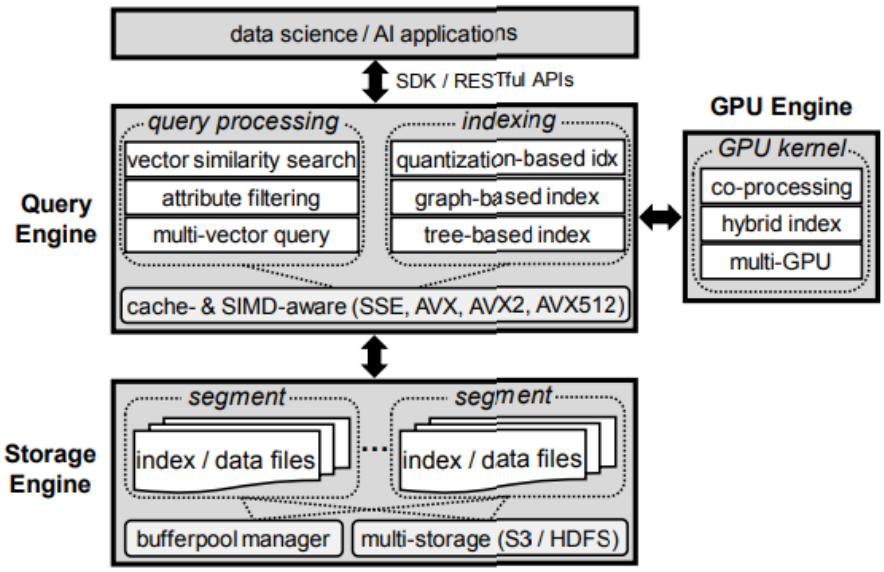


Figure 3.3: System architecture of a single instance of Milvus. Image from [WYG+21].

- Query engine: this component supports vector query processing and is optimized to run with new processors to reduce cache misses by exploiting SIMD instructions.
- GPU engine: it is a co-processing engine to the Query engine that accelerates performance through parallelism by taking advantage of the GPU.
- Storage engine: enables persistence and incorporates an LSM-based framework for dynamic data management. This system is supported by various file systems, in-

cluding Amazon S3 [Ama], local file systems and HDFS [SKRC10]. In the following paragraphs, we will describe the basic Milvus characteristics; for more in-depth knowledge we refer to the work by Guo et al. [GLX+22].

3.3.2 Features

We can distinguish various features in Milvus, the most interesting and useful for the purposes of the study are listed below:

- Query Processing: before defining how queries are processed we need to know the objects Milvus works on, i.e., entities. Milvus represents any object composed of vector data and, optionally, unstructured data as an entity. Once the entities have been saved in memory, it is possible to search by vector, attributes (i.e. by applying filters on the attributes of the entities) or by multi-vector (data aggregation). All of these queries support various similarity functions.

- Indexing: another factor of vital importance for the functioning of the system concerns the types of indexes supported as these affect the type of data structure used and the efficiency of the service. Referring to the latest studies [LZS+20] it was decided to support quantized indices, such as FAISS Product Quantization, Inverted Indexes and Graph-based Indexes. Nevertheless, Milvus provides interfaces for simple integration of new indexes.

- Dynamic data management: to support this functionality Milvus adopts the idea of LSM-trees [LC20]. Every time you want to insert a new vector in memory, you must wait for a pre-established timeout or for the achievement of a satisfactory size to move this set of vectors, called segment, to disk, trying to join it to other segments already present to ensure sequential access to secondary memory. Also, the segment, where both indexes and data are stored, is considered the basic unit for searching, scheduling and buffering.

- Storage Management: Milvus stores entities in memory via columns. The individual vectors are saved contiguously and, since each vector is of the same size, it is possible to directly access the portion of memory where the searched data resides via its id. In the case of multiple vectors, saving via column is applied.

  To make this memory management easier, we provide an example. Consider 3 entities A, B, C each containing two vectors $v1$ and $v2$, in memory data will be saved with this layout: $\{A_{v1}, B_{v1}, C_{v1}, A_{v2}, B_{v2}, C_{v2}\}$.

  The same method is used for the attributes, which are saved through columns. Each column is identified by $< key, value >$ where key is the attribute value and $value$ is the row id.

3.3.3 Optimizations

As already reported in subsection 3.3.2, Milvus uses the FAISS indexes, with some modifications to make it as efficient as possible. We can mainly denote 5 optimizations:

- Cache-aware design and multiprocessing: FAISS allows to use multiple threads to query the index, to do this the query vector array Q is divided by the number of threads, thus allowing to query the data structure in parallel. Milvus decides to change the approach, as shown in Figure 3.4, by assigning to each thread a portion of the vectors present in memory. Each thread will take all the query vectors as input and the search space will be restricted. Once the k elements resulting from the search have been found, a heap per thread and query vector will be created and, in a second phase, these heaps will be merged, thus obtaining the final result. With this type of dataset partition, Milvus tries to minimize the search space for each thread in order to load it completely in the cache, precisely in the L3 cache. In this way, the amount of cache misses is much lower than that of FAISS which, using a different approach, was therefore slower due to this problem.

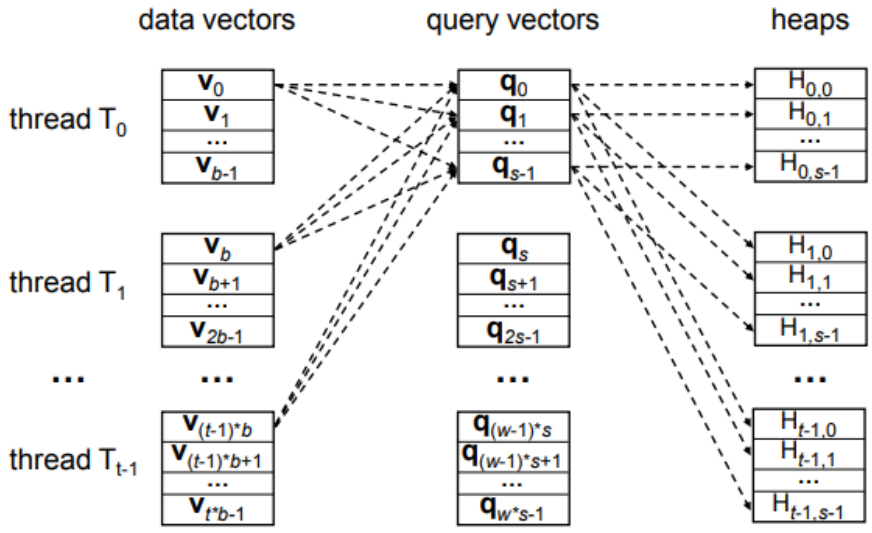


Figure 3.4: Milvus parallel computing and resource allocation. Image from [WYG+21].

- SIMD instructions: FAISS allows the use of SIMD instructions by specifying an attribute at compile time. Milvus, on the other hand, has refactored the entire FAISS codebase by implementing specific functions for the four types of supported SIMD instructions (SSE, AVX, AVX2, AVX512), by compiling them with the appropriate flag. In this way it is possible to choose the best instruction to execute directly at runtime.

- Bigger K neighbors with GPU: When operating on GPU, FAISS allows a maximum search of 1024 neighbors. This limit was enforced due to a physical limit of shared memory. Milvus overcomes this problem by executing incremental queries taking the maximum distance of the neighbors resulting from the previous query and executing a subsequent query considering only elements with greater distance than the previously retrieved maximum distance. In this way it is possible to chain an arbitrary number of queries, removing the previously imposed limitation.

- Multi-GPU support: FAISS allows compiling the code on an arbitrary number of GPUs (c) and, therefore, to execute this binary only on servers with a number of GPUs $>= c$. Milvus, through a segment-based scheduler, assigns a search task, or a segment, to each GPU. This type of scheduler removes the usage limit on the number of GPUs, allowing it not to specify the number of GPUs during compilation and, therefore, to run the binary code on servers with an arbitrary number of GPUs.

- CPU and GPU co-design: to perform queries on GPU a massive movement of data from the CPU is performed, causing a very expensive operation. To do this FAISS moves a single bucket at a time which does not fully use bandwidth, wasting it. This solution was chosen to avoid complex management due to missing buckets that have been previously deleted. Milvus, on the other hand, moves multiple buckets using full bandwidth and obviating deletions management issues by adopting an LSM-based out-of-place approach [LC20]. Furthermore, Milvus allows dynamic management of CPU and GPU by choosing which hardware component will be used based on the number of queries, as running on GPU may not be convenient due to the massive movement of data.

3.3.4 Distributed System

We can divide the architecture into many parts, as shown in Figure 3.5. The storage one is a Distributed shared storage that concerns data that is shared between the Milvus instances. In order to guarantee consistency Milvus adopts snapshot isolation, i.e., each query works on its own snapshot, taken at the moment of its first execution, at query time. The database updates do not interfere with queries that are already running. This kind of consistency policy was necessary because Milvus needs to support multiple access to its memory, in particular, Milvus is a read-intensive system, so it is necessary to have multiple readers while a single writer is sufficient. These worker instances are called worker nodes as shown in Figure 3.5. This type of architecture, based on a single writer, still satisfies the user's needs. The writer manages the operations of inserting, modifying and deleting from the database, while the readers process the user's queries, using as much as possible their internal memory, which is more efficient than storage, to decrease network traffic. Data is shared between readers via consistent hashing [KLL$^+$97], while sharding information is

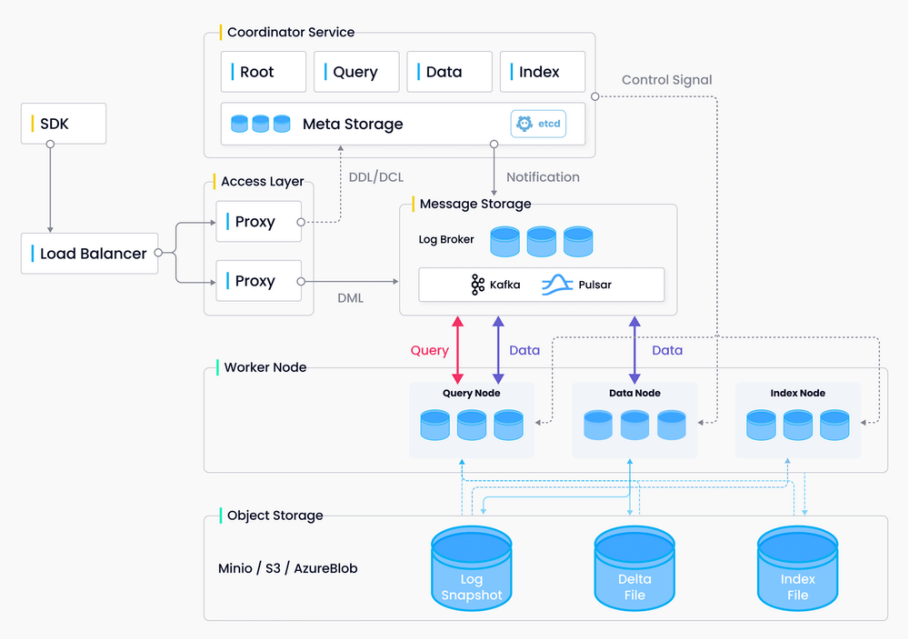


Figure 3.5: Milvus Distributed System. Image from [Milb].

kept within the Coordination layer. In this architecture the faults of the various instances are contemplated and managed. When a reader crashes it is simply pulled up again, while on a writer's crash, the system relies upon write-ahead logging [MHL+92] to ensure atomicity. These multiple instances need, as previously mentioned, a Coordination layer. This layer manages the coordination between reading and writing, load balancing and maintains metadata for the correct functioning of the system. This layer manages to have a high level of availability thanks to Zookeeper [HKJR19]. This architecture allows to reach a sub-linear scalability, as reposted in the experiments at section 7 of [WYG+21].

Chapter 4

Libraries Integration

To test the libraries presented in chapter 2 and chapter 3, some code was written, mainly in Python, to run them on the provided dataset. This code is free and open-source, available on Github [the]. In the following sections we will analyze the execution flow of each library, paying attention to the peculiarities of each one. The KNN library will not be reported, in our case we refer to the version of *scikit-learn* [skl] since, although it is present in the codebase, it is only used to evaluate the precision of approximate libraries. Additionally, unlike most other pieces of software evaluated in this thesis, it was easy to install and use. Unfortunately, this library suffers from a major problem that limits its use on very large datasets, i.e. it is not possible to add vectors to the index if it is already trained. This means that once we have saturated the RAM during the training phase it will no longer be possible to proceed with the insertion of other vectors, making the library completely unsuitable for use cases like ours.

4.1 Common Implementations

The presented repository provides an implementation for each library while respecting some common code portions. The most interesting aspects that deserve attention are the use of a *Chronometer* class since, in order to compare libraries, it is interesting to calculate the total time for creating an index and for searching within it, and how the dataset is read. Since the dataset is made up of 215 [data, datb] files in *csv* format, we need a reading pattern that allows us to create all the library indexes, paying attention to the vector insertion methods, which are different for each library. It was, therefore, chosen to leave the files separate and read them one at a time, considering the vectors as *numpy array* objects to save space in RAM, being lighter than native Python lists. The phase of inserting the vectors into the indices is governed by the size of the RAM. This is because

index data structures need to be trained on a portion of the dataset to be efficient. The larger the portion, the more precise the index will be. Once we have loaded as many as possible vectors into RAM it will be possible to perform training. This phase is called *training phase*. Once finished we will be able to continue with the insertion of the vectors into the index, but these will not shape the internal data structure, but only populate it, therefore not contributing to its refinement. We will continue with the insertion of the vectors until we have loaded the entire dataset.

4.2 Annoy

The Annoy library, which we presented from a theoretical point of view in section 2.1, is certainly the most user-friendly. It is very simple to install using *pip*, the Python installer package, and it allows us to immediately create an index. Within this index, we can insert all the vectors present in the dataset, one at a time. Annoy allows us to support vectors of type *float128* and therefore to use all the precision necessary to compare the vectors in the dataset. Once the insertion is complete we can build the index by providing the number of trees that it must use as a parameter. Obviously, the greater the number of trees, the greater the precision, but this will increase the time required for the index build. Unfortunately, this library does not provide the possibility to add vectors into the index after training it, this will lead to Annoy not being optimal on very large datasets, as we will see later. For the search we have only one parameter to tune, *search_k*, which controls the number of nodes that the search will check. By increasing this value we will have more precise results, but longer search times.

4.3 FAISS

FAISS, presented in section 2.2, is certainly the most complete library, as it provides many parameters and indexes that can be used. Installable via *conda*, a package manager for Python and other programming languages, it allows the creation of various types of indexes, even combined ones, as can be seen in the documentation [faij]. Once the index has been created we must insert vectors inside it, paying attention to the data type as FAISS does not support *float128*, instead it is possible to use *float32*, losing a little of precision. Since FAISS was designed to support billions of vectors, it is possible to insert a part of them, train the model and subsequently add other vectors. Obviously the greater the amount of data on which we train the index, the greater the precision. This trade-off is governed by the amount of RAM since training must be performed on a portion of data fully loaded into RAM, placing this as the upper limit on the number of vectors that can be used in this phase. Another peculiar aspect of this library is that to support cosine similarity, i.e. the

function for calculating the distance on which our dataset was encoded, we must manually normalize the vectors using a specific function.

This library aims to cover as many use cases as possible in terms of dataset size, accuracy and search time. To do this we can see a large section [Faid] of the wiki dedicated to the parameters necessary for creating the index. Going into the detail of the creation phase we have two patterns that we can follow. The first, more intuitive, consists of manually creating the indexes on the code side using the functions exposed by the library, passing the necessary parameters to them. This method is convenient to use only if we have to run on very small datasets using very simple indexes. The second method, which is more complex but also more powerful, consists in using the *index_factory*, i.e. a method that parses a string and, based on what has been inserted, generates the index, as discussed in [Faic]. The question may arise, why two types of index creation patterns? This was chosen because FAISS includes a disproportionate amount of parameters, indexes and additional components to use, which would not be easily usable without the help of *index_factory*. For our use case, indexes were created only with *index_factory*. This involved an in-depth study of all the parameters and components made available by the library, selecting the best ones by referring to the computing power available, presented in chapter 5. We can, therefore, schematize the creation of an index into five components, which will be presented in the actual parsing order of the string:

1. Prefixes: this phase is only useful for configuring the index to enable some special data insertion functions.

2. Vector transforms: in this phase we want to perform a transformation of the vectors before adding them to the index, reducing their dimensionality. The most common use involves using the Optimized Product Quantization [GHKS13] (using the Orthogonal Product Quantization Matrix) or the Principal Component Analysis Matrix to reduce the dimensionality of the input data.

3. Non-exhaustive search components: this is the phase where the indexes are chosen. The categories of indexes that FAISS provides are Inverted file indexes, Graph-based indexes and, for more complex cases where the datasets exceed one billion [Faif] or thousands of billions [Faih] vectors, Composite indexes [Faib].

4. Encodings: in this phase we will compress the vectors after they have been inserted into the index to save disk space and, in a subsequent loading of the index for a search, RAM space. The main encoding techniques concern: Product Quantization and Scalar Quantization.

5. Suffixes: here we define the refining techniques used to increase the precision of the index by reordering the results during the search phase through the exact computation of the distance from the query vector.

Obviously, each single component consists of a much greater number of solutions than those presented as well as being made up of a large quantity of parameters on which to perform tuning. Among these, we want to report the most impactful during the creation of the index. The first is certainly the order of magnitude that we want to use in phase 2, i.e. by how much we want to reduce the dimensionality of the vectors. Reducing the dimensionality will allow easier processing of the data as we will have a smaller and lighter vector, but reducing the dimensionality also makes us partially lose the information on the dimension that we are going to remove obtaining a resulting vector whose dimensions are a combination of the original ones. The second parameter is an index configuration parameter, in phase 3. In this case, the name of the parameter depends on the type of index we want to use, in the case of Inverted Index or Inverted Multi Index we refer to the number of centroids, while in the case of HNSW we refer to the number of edges that each single node has. These values change both efficiency and execution times. Finally, we have the compression parameter used in phase 4, which tells us how much to compress the vectors to save them in memory. This parameter will influence the quality and precision of the checks between individual elements in order to return the most correct possible result. Obviously, all these parameters must be chosen based on the dataset on which we want to execute, also paying attention to the RAM on which we load the index.

In conclusion, we want to provide an example of a string that can be used by *index_factory* with its explanation to make learning this creation pattern easier. Let's parse the string:

```
OPQ16_64,IVF262144(IVF512,PQ32x4fs,RFlat),PQ16x4fsr,Refine(OPQ56_112,PQ56)
```

- `OPQ16_64`: OPQ (Optimized Product Quantization [GHKS13]) pre-processing
- `IVF262144(IVF512,PQ32x4fs,RFlat)`: IVF index with 262k centroids. The coarse quantizer is an IVFPQFastScan index with an additional refinement step.
- `PQ16x4fsr`: the vectors are encoded with PQ fast-scan (which takes 16 * 4 / 8 = 8 bytes per vector)
- `Refine(OPQ56_112,PQ56)`: the re-ranking index is a PQ with OPQ pre-processing that occupies 56 bytes.

4.4 SPTAG

For this library, presented in section 2.3, the most convenient solution to install it is through the use of *docker*, a platform for building and running code within isolated containers. Like FAISS, this library does not support *float128*, only *float32* and expect an index training

step to create the correct data structure to contain all the vectors of the dataset, which can also be added at a later stage.

This library is the most complex, the least documented, and the one with the smallest community. Despite the huge codebase behind it, there is no documentation section, but there is only a *.ipynb* file with some examples. What has been studied and used in this library has been possible thanks to numerous tests and through reading the source code in C++. A peculiarity of this library, however, is that of being able to create the index and carry out searches without writing code, but simply passing all the necessary parameters to executables that are created during the installation of the library. This solution may seem very convenient, but unfortunately the executables only take as input certain types of data, formatted in a certain specific way, thus making them unusable by all those users who do not fall within their use case, therefore only allowing the use of the library by writing code.

4.5 Distributed FAISS

Distributed FAISS, presented in section 3.1, is nothing more than a library that acts as a facade for the FAISS implementation to allow us to use it in a distributed manner. The simplest and most recommended use of this library is through the *slurm* [slu] cluster and the *submitit* [sub] library, which allows the execution of Python code on multiple machines in the aforementioned cluster. Alternatively, you can also use the APIs made available by the library. To use the library it is sufficient to administer the indexes and add the various vectors via *round-robin* to ensure that each machine has a portion of the dataset. Data will then be automatically inserted into the machines, whose addresses will be provided via a specific file. What happens in the final machines is nothing other than the execution of FAISS. The results obtained from a search will be sorted to provide us with the best result and then returned to the client who made the request. However, these results may not be satisfactory if correct tuning of the parameters is not first performed. Unfortunately, this operation was not designed within the code available in the official repository. To overcome this problem I personally implemented the necessary functionality and opened a pull request to Meta to merge my code, currently awaiting approval [faik]. In addition to this implementation, it was also necessary to develop a method for inserting metadata since the library does not provide this automatically. This, however, was understood only after various tests and implementation trials as it was not reported in the documentation.

Unfortunately, this is not the only downside of this library as, although it may seem like a mature solution, it is not properly supported as very little documentation is written, there is no longer a development/support team behind it and many of the useful information for its use were obtained by reading the source code. Furthermore, it is correct to point

out that the use case of this library is limited to very few situations, for which it might be better to write your own solution, perhaps taking inspiration from this library. This is caused by the many missing features to obtain satisfactory results, implemented by us. This lack of uses is probably attributable to the great popularity, complexity and efficiency of the single-machine version of FAISS, which covers almost all the needs of users and companies without problems as you will see in chapter 5.

4.6 Distributed SPTAG

This library, presented in section 3.2, is the distributed version of SPTAG (section 2.3). The codebase from which the distributed version is taken is the same as the single machine version, therefore entailing the same positive and negative sides. Using this library involves defining two configuration files: *service.ini* and *Aggregator.ini*. These files, respectively passed to the *service* and *aggregator* executables, are used to configure the cluster on which we are going to execute. We will have to manually create the index to assign to each server, taking care to correctly partition the dataset. This also involved the implementation of our custom indexing via metadata algorithm, as it was not managed directly by the library. Once this is done, the various servers, governed by the service.ini file, will execute the requested queries and then return the results to the Aggregator, the machine to which we connect with the Client. This will merge the results and return the best candidates. Unfortunately, the implementation of the Aggregator provided is not as complete as one might hope. In fact, it was necessary to further implement a custom *Reduce* phase where, through a partial sort, only the K requested elements were returned.

4.7 Milvus

Milvus, presented in section 3.3 is the best documented library [milc] among all those analyzed. Its functioning is different from what the previous solutions have accustomed us to since this solution is not just a library for approximate search, but a complete vector database. During the index creation phase we will connect to the cluster, which must use Kubernetes [kub], and create a *collection*, a sort of partition identified by a primary key, this must contain various *field schema*, i.e. the types of data that we are going to load into the index. In our case, we will have two *field schema*, the one relating to the id of the image and the one relating to the vector that represents it. However, the vector fields cannot be read as a *float128* as the library does not support it, the default *float* data type of Milvus will be used instead. We will then add all the vectors of the dataset to the *collection*. Once this phase is completed, it will be sufficient to execute the command for the effective creation of the index and for its upload to the network. In the search phase

we will have to query the previously used *collection*.

Chapter 5

Experiments

In this chapter we test the libraries through the code presented in chapter 4 and available on Github [the], referring to the use-case provided by *Wikimedia Foundation* in chapter 1. We evaluate two approaches:

- Vertical scaling: in this case we run the tests on a single machine with a large amount of resources as you can see in Table 5.1. In addition to this, we were forced to add a second machine, much less performing, as can be seen in Table 5.2, for reasons that will be explained in section 5.3. From this moment on we will call the first "single powerful machine" and the second "single weak machine". Obviously, the use of this solution does not involve the execution of the libraries presented in chapter 3 as a single machine is used.

S.O.	Ubuntu 20.04.6 LTS
Kernel	Linux 5.15.0-73-generic
CPU	Intel(R) Core(TM) i9-10980XE CPU @ 3.00GHz
RAM	128 GB
Physical disk	SSD NVMe ∼3 TB

Table 5.1: Information of the single powerful machine used for vertical scaling

- Horizontal scaling: in this case we run the tests on a cluster made up of eight machines, individually less powerful than the aforementioned single powerful machine, as shown in Table 5.3. Obviously, the use of this solution does not involve the execution of the libraries presented in chapter 2 as they are not distributable.

We believe it is interesting to evaluate both solutions since each working group, evaluating

S.O.	CentOS Linux 7 (Core)
Kernel	Linux 3.10.0-1127.el7
CPU	Intel(R) Xeon(R) E5520 CPU @ 2.27GHz
RAM	12 GB
Physical disk	HDD ∼640GB

Table 5.2: Information of the single weak machine used for vertical scaling due to the malfunction "Memory corruption" reported in section 5.3

S.O.	CentOS Linux 7 (Core)
Kernel	Linux 3.10.0-1062.el7
CPU	Intel(R) Core(TM) i5-9500 CPU @ 3.00GHz
RAM	16 GB
Physical disk	SSD NVMe ∼256GB

Table 5.3: Information of an average machine used for horizontal scaling

its own internal availability and needs, could choose one scaling option rather than the other.

The tests we have carried out are performed only on CPU, proportionate to the type of algorithm used and to the reference use-case, as reported in Table 5.4. As you can see we have two tests on single machines, some on the single powerful machine and some on the single weak machine. Unfortunately, some problems with the single powerful machine did not allow us to carry out further tests on this machine in addition to those shown in the table. To overcome this, we have included further tests on the single weak machine, which we believe are very significant. Further details can be found in section 5.3.

Dataset	**Use-case**	**Libraries**
16 GB	Single powerful machine	Annoy, FAISS, SPTAG
32 GB	Single powerful machine	Annoy, FAISS, SPTAG
64 GB	Single powerful machine	Annoy, FAISS, SPTAG
Variable training (16 GB)	Single weak machine	FAISS, SPTAG
Full dataset (215 GB)	Single weak machine	FAISS
128 GB	Distributed machines	Distributed-FAISS, Distributed-SPTAG
Full dataset (215 GB)	Distributed machines	Distributed-FAISS, Distributed-SPTAG

Table 5.4: Test suite

As you can see in Table 5.4 we have inserted a test case called "Variable training (16 GB)". It was inserted due to some problems reported in section 5.3 and consists of varying the

portion of the dataset provided to the libraries during the training phase and adding the remaining vectors later. This test is performed on the single weak machine and follows the test cases reported in Table 5.5. In this way we are able to evaluate which library performs best when the size of the dataset in the training phase varies. This is important because in a real case we will rarely attempt to create indexes trained on 100% of the dataset, but we will always have limitations provided by the amount of RAM, which, in some cases, has a capacity much lower than the size of the dataset.

Dataset	**Portion used in training phase**	**Libraries**
16 GB	100%	FAISS, SPTAG
16 GB	75%	FAISS, SPTAG
16 GB	50%	FAISS, SPTAG
16 GB	25%	FAISS, SPTAG

Table 5.5: Test suite for "Variable training" on single weak machine

For each test we have analyzed the times needed to build the index and the times needed to search. The latter one is computed as an average time between all the searches, five on different input vectors for each index. Once we have obtained the output from each library we analyzed the precision of the algorithm. To obtain this value we have to run the K-Nearest Neighbors (KNN), which carries out a complete, precise and non-approximate search, and compare the results of the libraries with those obtained from the aforementioned method. The searches require 100 neighbors and the precision is computed by counting how many elements, of the 100 returned, are placed among the top 100 elements returned by the KNN. We consider a value as precise only if it is one of the 100 nearest neighbors. However, remember that both the timing and precision are subject to variations that depend on some parameters that are provided to the index. We also want to inform that, during the parameter tuning phase, we paid more attention to search efficiency (time and precision) rather than low index creation time. However, we carried out significant parameter tuning, without going into the details of the ad-hoc compilation for the CPU. The desirable goal is to have the highest possible precision and at the same time have searches that take as much time as the average latency of a request on the network, estimated to be between 20 and 100 milliseconds.

5.1 Dataset

The dataset on which we carried out the tests was provided by *Wikimedia Foundation*. This dataset contains 6.4 Million images, of different sizes and representing different subjects/objects. We have two types of the same dataset available, the one containing the

compressed images [data] and the one containing the vector representation of these images [datb], both have already been shuffled. We used the latter, which was created using cosine distance.

The dataset is made up of 215 files, each approximately 1 GB. Each of these contains 30122 rows and 2048 columns. Each element of each vector is represented with the data type *numpy.float128*.

5.2 Parameters Tuning

This section is necessary to have correct, efficient and precise functioning of our indices when we are going to execute on very large datasets. We will therefore analyze which parameters have been tuned to carry out the tests reported above. The values we obtained were found based on empirical tests and theoretical studies of the functioning of the internal components of the indexes.

- Annoy: for this library, we only had to tune two parameters: *n_trees* and *searck_k*. The first indicates how many trees should be created from the index. A greater number of trees increases precision, but also the times for creating the index and the memory that it occupies. The second parameter tells us how many tree nodes will be checked. Higher values indicate more precise, but slower searches.

- FAISS and Distributed-FAISS: as regards this library, the number of parameters to be tuned is enormous. As we reported in section 4.3 and in the official wiki [Faic]. The main ones are dimensionality reduction, index feature and vector compression. As regards the first parameter, we carried out tuning only in our last test case, i.e. section 5.8, in which we used the OPQ (Optimized Product Quantization [GHKS13]) component. This component requires two parameters that tell us respectively how many subvectors and how many dimensions we want to obtain. Next, we have the parameters relating to the characteristics of the index. These parameters therefore depend on the type of index we want to use. In the case of Inverted File Index we noticed how a value between 4 and 16 centroids was optimal for 1 GB of our dataset. Using this information it was easy to find the number of centroids for larger portions of the dataset. The same reasoning was made regarding Inverted Multi-Index but through a different formula. The parameter that is passed governs the number of centroids as follows $centroids = 2^{2*param}$. We noticed that for 1 GB of dataset the optimal value was 1. In this case the increase of this value must be more parsimonious as the number of centroids increases exponentially. We therefore decided to choose this value using the formula $param = \log_2(n) + 1$ where n is the size of the dataset. In the case of HNSW, however, the parameter to pass is related

to the number of arcs that each node of the graph must have. The greater the number of arcs, the greater the precision of the index at the expense of its size. We managed to always keep this parameter between 32 and 64, recommended by the documentation as the optimal values. Finally, we have the parameter to configure the vector compression phase for saving vectors in memory. Here too we have many possibilities, in our case we always use Product Quantization in which we specify the number of blocks that will be encoded with 4 bits according to the "fast scan" method which uses SIMD instructions to calculate the distance. We also carried out Refinement phases in almost all executions to improve the precision of the index. This was possible via RFlat and Refine method (which accepts other parameters). In addition to these parameters necessary during the creation of the index we also used some others to improve the precision of the search phase as a trade-off between precision and execution time, as it can be read in [Faie]. The most important are: *nprobe*, *k_factor_rf* and *efSearch*. The first two are used by Inverted File Index and Inverted Multi-Index. The first one, *nprobe*, indicates how many cells of the index must be checked during the search phase, while the second indicates how many must be reordered among all those returned from the checked cells. The last parameter, *efSearch*, is used only by HNSW and indicates the number of graph nodes to visit. All these parameters are necessary for both the FAISS library and its distributed version. Unfortunately, however, in the distributed version the parameters that can be used in the search phase are not implemented natively and it is therefore necessary to use the Pull Request containing my implementation of this functionality [faik].

- SPTAG and Distributed-SPTAG: regarding SPTAG there is no satisfactory documentation to fully understand which parameters to use and how to do it. We have noticed how for the BKT index we can avoid carrying out parameter tuning as, thanks to the repeated refinement phases, the index is already very precise. This, however, is not true for the SPANN index which is completely devoid of documentation even if, through an example reported in a *.ipynb* file, it was possible to know some parameters that we tuned to improve its performance in a completely empirical way without having support from the library. Since SPTAG and Distributed-SPTAG are part of the same codebase, everything we have just reported is also applicable to the distributed version.

5.3 Limitations and Malfunctions

Unfortunately, during the creation of the various environments and the testing phase, we encountered problems that did not allow us to continue with the desired plan, leading to delays, changes and alternative solutions. Below we listed the most significant problems we encountered, what caused them and the countermeasures we took.

- Milvus with Kubernetes: while studying the library and documentation we learned that the library was mature enough to be used in production environments for docker clusters and Kubernetes. Unfortunately, in the subsequent phase we realized that the docker cluster is not yet supported for carrying out distributed searches, but only for standalone test environments and, therefore, single machines. This led to slowdowns due to the study of other possible solutions as the single machine version would not have made sense as it is against nature for this library and we do not have a Kubernetes cluster available. Since the *Wikimedia Foundation* infrastructure currently does not support Kubernetes, we have chosen to run all the other pre-established tests on all the other libraries and to evaluate the purchase of a Kubernetes cluster only if the efficiency of the other solutions were extremely low, also remembering that Milvus is based on an improved version of FAISS making us expect similar results. This solution seemed the most sensible to us in relation to the needs of the *Wikimedia Foundation* and the use case provided.

- Memory corruption: after many tests performed on the single powerful machine reported in Table 5.1, it suffered some problems with the storage memory which, in a non-deterministic manner, made many files in the dataset corrupt without the possibility of correcting them. One solution we tried was to use different backups and different partitions, but it was still not possible to run the algorithms as the corruption continued to occur. The missing tests on a single machine are the tests relating to portions of the 128 GB and 215 GB dataset (full dataset), these tests would have been very interesting as we would have had to train the indexes on small partitions of the dataset since the RAM would not have been enough to load it entirely for the training phase. To overcome this problem we decided to add the test suite reported in Table 5.5 in order to, combining the results already obtained from the other tests in Table 5.4, predict the execution times and accuracy for the missing tests in an approximate and heuristic manner. Since we consider very interesting to run the libraries on the entire dataset despite all the limitations and malfunctions we encountered, we decided to carry out another additional test, reported in Table 5.4 as "Full dataset (215 GB) - single weak machine" on the machine reported in Table 5.2. This test provides us with the experimental comparison to what we have hypothesized through other tests although on an extremely less powerful machine.

- Distributed SPTAG with SPANN index: within the material studied for this library [CWL+18] it is reported that SPANN and, in general, SPTAG are technologies mature enough to be distributed. Unfortunately, this is not the case, as also reported in chapter 4 for the SPANN index. The SPTAG documentation, despite being almost non-existent, does not provide the material and examples necessary for the correct understanding of the use of the SPANN index in a distributed manner. Despite this major limitation, the C++ source code was read trying to obtain a solution, but unfortunately it would seem that this index is not supported in a distributed manner.

To overcome this problem we wanted to replace the distributed tests of SPANN with those of BKT in order to still have a distributed SPTAG implementation.

- Machine's ownership and execution times: all the machines we use are not owned by individuals but by research groups or university courses. This led to some conflicts during the installation of the necessary technology stacks, but above all this interfered with the execution times of the libraries and individual indexes. By paying attention when performing searches to detect the average execution time, we managed to obtain clean values, which do not suffer from the problem reported above. Unfortunately, it was not possible to do the same with the index creation times. This is because these times, unlike searches, are much longer, even by several hours. We would therefore like to point out that all index creation times may be slightly longer than they would be if we had full access to the machines. However, we believe that this does not completely invalidate our measurements as we are interested in the order of magnitude of these execution times, which remains true despite this disturbance.

- Internet network and power grid problems: we also faced problems that did not depend on the technologies or machines we ran on, such as Internet network and power grid problems. The machines on which we have performed do not belong to us and, therefore, we have no possibility of physically visiting them to carry out operations on site. This means that we are dependent on the network in which the machines reside. Unfortunately, it happened several times that we were unable to reach the machines for weeks and, initially, even for months. We also encountered some power grid problems which led to the machines being turned off and therefore not being able to be used for several days. To overcome this discontinuity in use, we have increased the number of local tests in order to bring onto the machines only code on which we are sure of obtaining satisfactory results.

5.4 Single Machine on 16 GB Dataset

With this test case we want to show which library is the best to use if we have a dataset of around 16 GB and a sufficient amount of RAM to be able to train the indexes on the entire dataset. The measurements were taken referring to what is presented in chapter 5. We also remind that the technical specifications of the indices used are presented in the footnotes.

Library	**Index type**	**Creation time**	**Search avg time**	**Average accuracy**
Annoy	Random Projection Trees[1]	820 s	69 ms	95.2 %
FAISS	Inverted Index[2]	26 s	13 ms	99.6 %
FAISS	Inverted Multi Index[3]	25 s	14 ms	99.8 %
FAISS	HNSW[4]	87 s	8 ms	99.4 %
SPTAG	BKT[5]	8790 s	8 ms	99.4 %
SPTAG	SPANN[6]	6627 s	4 ms	97.6 %

Table 5.6: Index creation time, Search average time and Average accuracy for each index in section 5.4

As we can see from Table 5.6, all the indices allow us to have very high precision, this is certainly because the entire dataset was used in the training phase. We can also be satisfied with the execution times as they are in the order of milliseconds, excellent for our use case. However, we have two indexes whose creation is extremely slower than the others, namely SPTAG. This is due to the fact that this library carries out self-tuning phases to improve itself as much as possible in each adding phase. Obviously, this involves longer creation times, but also less effort and study in the search for the best parameters, which, even if provided, give details on the structure of the index, leaving the precision and efficiency dependent on the number of refinement iterations. Although this execution makes us very satisfied we can start to notice which index is probably not suitable for too high workloads, namely Annoy.

[1] 64 trees, search_k=32768

[2] IVF256,PQ2048x4fs,RFlat, nprobe=64, k_factor_rf=3

[3] IMI2x5,PQ2048x4fs,RFlat, nprobe=192, k_factor_rf=4

[4] HNSW64, efSearch=256

[5] Balanced k-means tree and relative neighborhood graph, 32 Threads

[6] SPANN, 32 Threads, Base: {IndexAlgoType: BKT}, SelectHead: {isExecute: True}, BuildHead: {isExecute: True, RefineIterations: 3}, BuildSSDIndex: {isExecute: True, BuildSsdIndex: True, PostingPageLimit: 12, SearchPostingPageLimit: 12, InternalResultNum: 32, SearchInternalResultNum: 64}

5.5 Single Machine on 32 GB Dataset

This test case is no different in terms of methods and purpose compared to the previous one, the difference lies in the size of the dataset provided. We therefore want to understand which library is best when running on a dataset of approximately 32 GB, which is used entirely in the training phase. The measurements were taken referring to what is presented in chapter 5. We also remind you that the technical specifications of the indices used are presented in the footnotes.

Library	Index type	Creation time	Search avg time	Average accuracy
Annoy	Random Projection Trees[7]	1828 s	79 ms	92.6 %
FAISS	Inverted Index[8]	39 s	13 ms	98.6 %
FAISS	Inverted Multi Index[9]	33 s	11 ms	96.8 %
FAISS	HNSW[10]	193 s	12 ms	98.6 %
SPTAG	BKT[11]	18425 s	7 ms	99.6 %
SPTAG	SPANN[12]	14073 s	5 ms	97.2 %

Table 5.7: Index creation time, Search average time and Average accuracy for each index in section 5.5

Although the dataset used is twice the size of the previous test, we have no news from the point of view of the results. All index construction times and some search times have certainly increased as shown in Table 5.7, but despite this we still manage to have extremely satisfactory results with excellent precision. We can, however, confirm the hypothesis that we had already posed in section 5.4, that Annoy is the library least suited to running on large datasets.

[7] 64 trees, search_k=32768

[8] IVF512,PQ2048x4fs,RFlat, nprobe=64, k_factor_rf=3

[9] IMI2x6,PQ2048x4fs,RFlat, nprobe=192, k_factor_rf=4

[10] HNSW64, efSearch=256

[11] Balanced k-means tree and relative neighborhood graph, 32 Threads

[12] SPANN, 32 Threads, Base: {IndexAlgoType: BKT}, SelectHead: {isExecute: True}, BuildHead: {isExecute: True, RefineIterations: 3}, BuildSSDIndex: {isExecute: True, BuildSsdIndex: True, PostingPageLimit: 12, SearchPostingPageLimit: 12, InternalResultNum: 32, SearchInternalResultNum: 64}

5.6 Single Machine on 64 GB Dataset

This test case is the last one, in terms of dataset size, in which we can create indexes and train them on almost the entire dataset. The measurements were taken referring to what is presented in chapter 5. As the size of the dataset increases, we come up against the physical limit of the machine on which we are running the tests. Due to this, not all indices were trained on 100% of the dataset, but on a smaller portion. More precise information about these are presented in footnotes.

Library	**Index type**	**Creation time**	**Search avg time**	**Average accuracy**
Annoy	Random Projection Trees[13]	4040 s	101 ms	87.4 %
FAISS	Inverted Index[14]	139 s	14 ms	99.6 %
FAISS	Inverted Multi Index[15]	89 s	12 ms	97.8 %
FAISS	HNSW[16]	1488 s	16 ms	98.0 %
SPTAG	BKT[17]	34796 s	8 ms	99.0 %
SPTAG	SPANN[18]	43960 s	5 ms	82.4 %

Table 5.8: Index creation time, Search average time and Average accuracy for each index in section 5.6

Also in this case we do not have major differences with the previous test. Although some indices have been trained on a portion of the dataset and not on the entirety, we still have very high precision and excellent execution times, see Table 5.8. We can, however, officially exclude Annoy from future tests as it is definitely the worst library from the efficiency and accuracy point of view. Furthermore, this library does not allow training the index on a portion of the dataset and then continuing to add data, thus making it unsuitable for datasets so large that they cannot be fully loaded into RAM. However, we note that, despite very low search times and excellent precision, SPTAG takes disproportionately long times to build the index. Referring to Table 5.8 we can read 34796 seconds for BKT, which is more than 9 hours, and 43960 for SPANN, which is more than 12 hours. This means that this library, despite its performance, may not be the best in all use cases, especially with even larger datasets. Furthermore, we can see that, in addition to the index creation time issue, SPANN also performed worse regarding accuracy. Although it is still the fastest in terms of search, this does not compensate for a precision that is almost 20% lower than

BKT, making it, in fact, uncompetitive.

[13]64 trees, search_k=32768

[14]IVF1024,PQ512x4fs,RFlat, trained on 93.75% (60 GB) of dataset, nprobe=64, k_factor_rf=3

[15]IMI2x7,PQ2048x4fs,RFlat, trained on 93.75% (60 GB) of dataset, nprobe=256, k_factor_rf=4

[16]HNSW64

[17]Balanced k-means tree and relative neighborhood graph, 32 Threads, trained on 93.75% (60 GB) of dataset

[18]SPANN, 32 Threads, trained on 89% (57 GB) of dataset, Base: {IndexAlgoType: BKT}, SelectHead: {isExecute: True}, BuildHead: {isExecute: True, RefineIterations: 6}, BuildSSDIndex: {isExecute: True, BuildSsdIndex: True, PostingPageLimit: 24, SearchPostingPageLimit: 24, InternalResultNum: 64, SearchInternalResultNum: 128}

5.7 Single Machine with Variable Training Size on 16 GB Dataset

In this section we want to see how the various libraries behave when the size of the dataset provided in the training phase on a single machine varies. This test is substantial as these libraries were created to support several millions if not billions of vectors, therefore using datasets so large that they cannot be completely loaded into RAM. The test case involves the comparison of different indices which will use 25%, 50%, 75% and 100% of the 16 GB dataset in the training phase. As you will see not all the indexes previously proposed are suitable for a partial training phase, narrowing the field to those present in Table 5.9. Also in this case technical specifications of the indices are presented in footnotes.

Library	**Index type**	**Creation time**	**Search avg time**	**Average accuracy**
	Trained on 100% of the dataset			
FAISS	Inverted Index[19]	26 s	13 ms	99.6 %
FAISS	Inverted Multi Index[20]	25 s	14 ms	99.8 %
SPTAG	BKT[21]	8790 s	8 ms	99.4 %
SPTAG	SPANN[22]	6627 s	4 ms	97.6 %
	Trained on 75% of the dataset			
FAISS	Inverted Index	46 s	12 ms	99.0 %
FAISS	Inverted Multi Index	25 s	15 ms	99.6 %
SPTAG	BKT	19539 s	6 ms	43.6 %
SPTAG	SPANN	4804 s	5 ms	97.6 %
	Trained on 50% of the dataset			
FAISS	Inverted Index	82 s	12 ms	99.2 %
FAISS	Inverted Multi Index	69 s	14 ms	99.8 %
SPTAG	BKT	13616 s	6 ms	34.4 %
SPTAG	SPANN	3029 s	4 ms	51.6 %
	Trained on 25% of the dataset			
FAISS	Inverted Index	109 s	12 ms	99.4 %
FAISS	Inverted Multi Index	46 s	12 ms	94.6 %
SPTAG	BKT	8123 s	6 ms	20.0 %
SPTAG	SPANN	1360 s	4 ms	26.4 %

Table 5.9: Index creation time, Search average time and average accuracy for each index in section 5.7

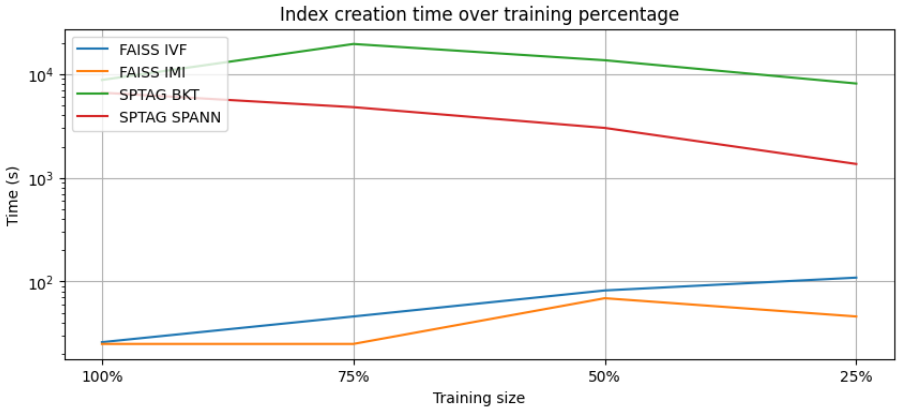
We can divide the analysis of this test case into three sections: creation time, search time and accuracy as shown in Table 5.9. As regards the first, we can see how the percentage of the dataset provided in the training phase certainly involves a change in timing, but not always in a predictable or monotonous manner. This means that the training phase is very dependent on the internal functioning of the algorithm, which is non-deterministic, and on noise caused by other processes performed by other users as reported in section 5.3. We can, however, note that the SPTAG indices have creation times hundreds, if not thousands, times longer than the FAISS indices; we can also notice that as the training portion decreases, SPTAG times decrease, while FAISS times increase. This is easily visible in Figure 5.1a. For the second case study, the one relating to search time, we can see how the percentage of datasets on which we train is not relevant, as reported in Figure 5.1b. This is an excellent point in favor of these algorithms because it will allow us to always have quick answers even with huge datasets on devices with limited RAM. Obviously, this data must also be compared with the precision of the search, which we can see is always optimal for the FAISS indices, unlike SPTAG, as easily shown Figure 5.1c. This leads us to conclude that, in case we had a very large dataset and/or little RAM to perform the training phase, the best solutions from every point of view are those proposed by FAISS, in particular the Inverted File Index.

[19]IVF256,PQ2048x4fs,RFlat, nprobe=64, k_factor_rf=3

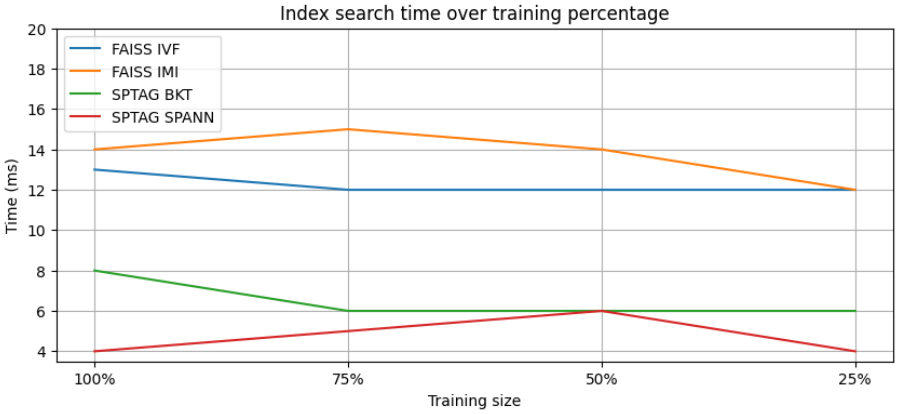
[20]IMI2x5,PQ2048x4fs,RFlat, nprobe=192, k_factor_rf=4

[21]Balanced k-means tree and relative neighborhood graph, 32 Threads

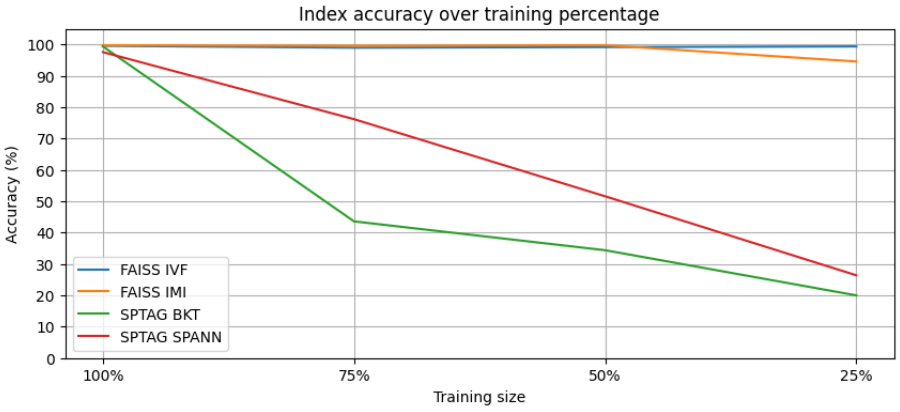
[22]SPANN, 32 Threads, Base: {IndexAlgoType: BKT}, SelectHead: {isExecute: True}, BuildHead: {isExecute: True, RefineIterations: 3}, BuildSSDIndex: {isExecute: True, BuildSsdIndex: True, PostingPageLimit: 12, SearchPostingPageLimit: 12, InternalResultNum: 32, SearchInternalResultNum: 64}



(a) Creation time changes versus the portion of the dataset used in the training phase



(b) Search time changes versus the portion of the dataset used in the training phase



(c) Accuracy changes versus the portion of the dataset used in the training phase

Figure 5.1: Graphical display of the data reported in Table 5.9

5.8 Single Machine on 215 GB Dataset

This test case was inserted following the single powerful machine malfunction reported in section 5.3. The idea behind this test is to perform an approximate search on the entire dataset on a machine with very limited resources, which is not able to completely load the dataset into RAM making it necessary to carry out the training phase on an extremely small portion of the dataset. This involves the exclusion of Annoy from the tests, as already announced in chapter 4, since it is not able to support a partial training phase. The machine on which we will run this test, reported in Table 5.2, has only 12 GB of RAM. Through some empirical tests we were able to notice that the training phase on it is able to use only 8 GB, or approximately 3.72% of the entire dataset.

The tests carried out and the results obtained are reported in Table 5.10. Unlike the other tests performed, we can note that we will only operate on the FAISS library. This is because according to what we have seen in previous tests, especially thanks to section 5.7, this library is the only one capable of carrying out fairly precise searches on indexes trained on very small portions of the dataset. We will therefore explore different FAISS indexes, focusing on those recommended by the documentation [Faii, Faif] for this particular use case. We also remind you that the technical specifications of the indices used will be present in the footnotes remembering that, as already specified in chapter 5, we have carried out a tuning of the parameters giving greater importance to the accuracy and search time (which we want to keep ≤ 100 ms) compared to the time necessary to create the index.

Library	**Index type**	**Creation time**	**Search avg time**	**Average accuracy**
FAISS	IVF (1)[23]	7459 s	89 ms	60.8 %
FAISS	IVF (2)[24]	31339 s	86 ms	63.4 %
FAISS	IVF + HNSW (1)[25]	7601 s	100 ms	60.8 %
FAISS	IVF + HNSW (2)[26]	9801 s	87 ms	61.6 %
FAISS	OPQ + IVF + HNSW [27]	34899 s	95 ms	67.0 %
FAISS	OPQ + IVF + HNSW + Refine[28]	18419 s	87 ms	64.6 %

Table 5.10: Index creation time, Search average time and average accuracy for each index in section 5.8

As we can see from the results presented in Table 5.10 and shown in Figure 5.2 it is possible to have decent precision, despite such a small training size. Obviously, it was necessary to carry out more demanding tuning to achieve these precisions, at the expense of execution time. We can say that, however, it will be very difficult to do better than what is shown in the table. We can understand this by looking at the type of indices used. Our idea was to follow three main paths: Inverted Index, Composite Index and Composite

Index with compression. The first solution is now well known and we know that it works efficiently even with very small training portions as reported in section 5.7. The second index family, i.e. the composite one, allowed us to have a second parameter to tune, used by the sub-quantizer, but it did not provide us with better results than the single Inverted Index. The last family is the one recommended by the documentation [Faif] and involves the use of composite indexes by first performing a compression on the dimensions. Since this index is composed of lighter vectors and, therefore, allows faster searches, it allowed us to increase the value of the *nprobe* parameter, which increases the precision at the expense of execution time, as can be seen in Figure 5.2. This solution provides us with slightly more precise results than other index families. Finally, we also wanted to try a refinement function, which however did not provide great improvements. We can finally say that, by training the index on such a small portion of the dataset, it will be extremely difficult to have precision greater than 70% while maintaining execution times lower than 100 ms. This allows us to conclude that if we wanted more efficient searches in terms of precision and execution time on a single machine it is necessary to upgrade the hardware to be able to train on a larger portion of the dataset.

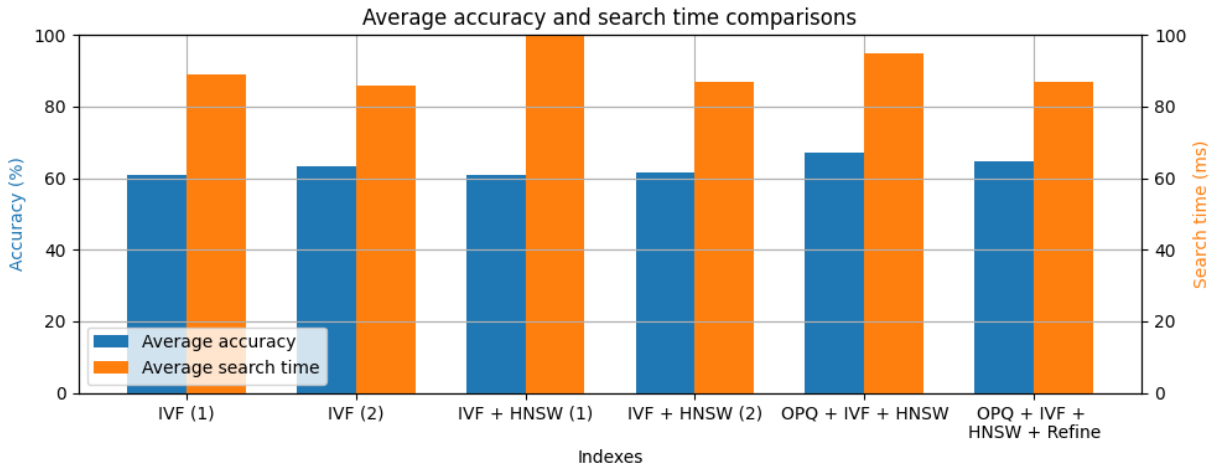


Figure 5.2: Comparison between accuracy and search time among the indices presented in this test case. The blue column refers to the left axis, the orange column to the right one. Data from Table 5.10

[23]IVF3340,PQ1024x4fsr, trained on 3.72% (8 GB) of dataset, nprobe=8

[24]IVF4096,PQ1024x4fsr, trained on 3.72% (8 GB) of dataset, nprobe=8

[25]IVF3340_HNSW64,PQ1024x4fsr, trained on 3.72% (8 GB) of dataset, nprobe=9

[26]IVF6120_HNSW64,PQ1024x4fsr, trained on 3.72% (8 GB) of dataset, nprobe=12

[27]OPQ512_1024,IVF4096_HNSW32,PQ1024x4fs, trained on 3.72% (8 GB) of dataset, nprobe=20, quantizer_efSearch=256

[28]OPQ256_512,IVF4096_HNSW64,PQ512x4fs,Refine(PCA72,SQ6), trained on 3.72% (8 GB) of dataset, nprobe=20

5.9 Distributed Machines on 128 GB Dataset

In this test case we will run FAISS and SPTAG in a distributed environment, on a cluster of machines, as presented in chapter 5. The peculiarity of running on a cluster is that of being able to distribute the workload and the dataset in order to mitigate the limit imposed by the RAM on a single machine. This led us to train and run both libraries on the entire dataset, corresponding to 16 GB for each machine, thanks to efficient management of the memory by our integrations and the libraries. Unfortunately, as we can see in Table 5.11, the SPANN index is not present for Distributed-SPTAG. This, as reported in section 5.3, is due to the fact that it is not distributable by the library as it lacks documentation and, probably, also the implementation to do so. We remind you that more details on the indices are available in the footnotes.

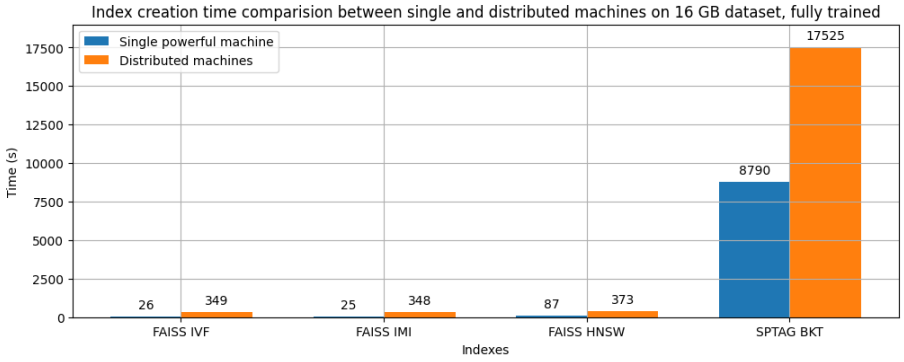
Library	**Index type**	**Creation time**	**Search avg time**	**Average accuracy**
Distributed-FAISS	Inverted Index[29]	2797 s	27 ms	100.0 %
Distributed-FAISS	Inverted Multi Index[30]	2789 s	14 ms	99.8 %
Distributed-FAISS	HNSW[31]	2986 s	17 ms	98.8 %
Distributed-SPTAG	BKT[32]	17525 s	28 ms	100.0 %

Table 5.11: Index creation time, Search average time and Average accuracy for each index in section 5.9

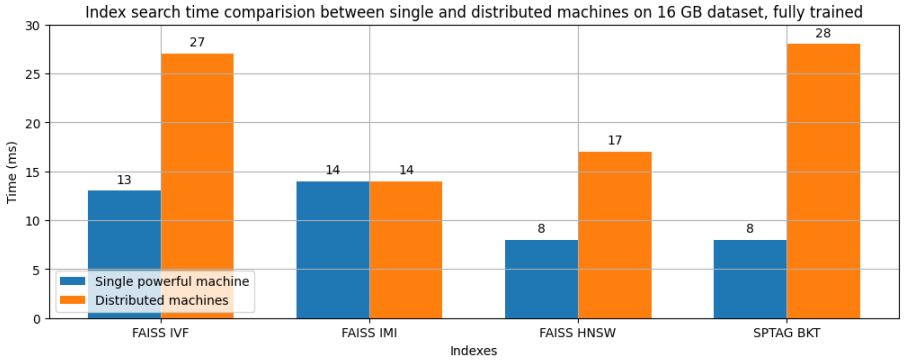
As you can see in Table 5.11 we obtained amazing results for both libraries. For Distributed-FAISS that was possible thanks to our pull request [faik] with which we managed to tune the *efSearch* and *nprobe* parameters. In this test case each node created its own index on 16 GB of the dataset. This same behavior has already occurred in section 5.4. We can then compare these two executions to see if the index creation time or search time has changed by having to go through a network of nodes rather than running directly on a single machine as shown in Figure 5.3. To carry out this comparison, however, we must first consider that the creation time of the index present in table Table 5.11 for Distributed-SPTAG represents a completely parallelized creation, while for Distributed-FAISS we have some operations performed sequentially. To correctly represent the times of this last library we divided them by the number of nodes in the cluster, i.e. 8. We can see in Figure 5.3a how both libraries have significantly longer creation times. This is certainly due to a difference between the computing power of the machines, even if the order of magnitude that passes between the two test cases is caused by other important factors. As regards Distributed-FAISS, this is due to the latency of transmitting the datasets from the master machine to the nodes where the index is contained. As far as Distributed-SPTAG is concerned, the impact of a less performing machine is greater, although from the point of view of the index structure this is caused by complex management of metadata which requires an additional internal

data structure. The metadata was necessary because Distributed-SPTAG does not allow the creation of indexes via a master machine, but must be created one at a time within the nodes, as explained in chapter 4. This means that, by accessing their portions of the dataset, each node will contain the sequential identifiers of its vectors, without considering the other nodes. This increases the complexity of the search since from the master node we cannot know if a certain ID is returned from one machine rather than another. This does not allow us to correctly map the vector ID to its image. To overcome this, each index contains the metadata to carry out this check internally, however slowing down creation and search times. As regards search times, reported in Figure 5.3b, these are longer due to the latency of the network during communication between the master and the nodes. Despite this, both libraries performed extremely well in both search speed and accuracy.

[29]IVF256,PQ2048x4fs,RFlat, nprobe=64
[30]IMI2x5,PQ2048x4fs,RFlat, nprobe=64
[31]HNSW64, efSearch=64
[32]Balanced k-means tree and relative neighborhood graph, 8 Threads



(a) Creation time changes between single machine and distributed machines. Time values of Distributed-FAISS have been divided by the number of machines present in the cluster as they represent sequential vector insertion



(b) Search time changes between single machine and distributed machines

Figure 5.3: Comparison of execution times between single powerful machine and distributed machines. Both solutions analyzed have only indices containing 16 GB of datasets, with the difference that the orange columns show the times in a distributed context, which suffers network latency and additional implementations necessary for communication between master and nodes. Data from Table 5.6 and Table 5.11

5.10 Distributed Machines on 215 GB Dataset

This further distributed test case is identical to section 5.9, with the difference that we are going to run it on a larger dataset. In this case we have 215 GB which is divided into portions of approximately 27 GB per cluster machine. Despite this, we still managed to carry out a training phase on the entire dataset thanks to the efficient use of memory by FAISS, while for SPTAG we were forced to train it on 16 GB per machine, carrying out total training on 60% of the dataset. Further information on the indexes and parameters used are presented in the footnotes.

Library	**Index type**	**Creation time**	**Search avg time**	**Average accuracy**
Distributed-FAISS	Inverted Index[33]	4759 s	48 ms	99.6 %
Distributed-FAISS	Inverted Multi Index[34]	4754 s	31 ms	100.0 %
Distributed-FAISS	HNSW[35]	5100 s	16 ms	98.0 %
Distributed-SPTAG	BKT[36]	19876 s	27 ms	99.8 %

Table 5.12: Index creation time, Search average time and Average accuracy for each index in section 5.10

As we can see from the results reported in Table 5.12, we don't have big differences compared to those reported in the test case in section 5.9. This is certainly due to the fact that the dataset for each single machine has gone from 16 GB to 27 GB, an increase which, as we had already tested on a single machine in section 5.4, section 5.5 and section 5.6, involves small and easily predictable changes in both execution time and precision. However, we obtained an excellent result from SPTAG, which underwent a training phase of 60% of the dataset, even though we have previously noted in section 5.7 that it is not very performing on small portions of training. Each machine on which the search was performed probably obtained a fairly low precision, but by combining the 100 results per machine and sorting among them it was possible to return all, or almost all, the 100 nearest neighbors, bringing the precision to be very close to 100%. We can therefore say that the conclusions to be drawn in this test case are the same as those provided on the previous distributed test over 128 GB reported in section 5.9.

[33]IVF430,PQ2048x4fs,RFlat, nprobe=64

[34]IMI2x5,PQ2048x4fs,RFlat, nprobe=64

[35]HNSW64, efSearch=64

[36]Balanced k-means tree and relative neighborhood graph, 8 Threads, trained on 60% (128 GB, 16 GB for each machine) of dataset

5.11 Discussion

In this section we will carry out an overall evaluation of all the tests previously reported and draw conclusions applicable to our use case. To have more details on the results obtained from the tests you can view the document which contains such information, published in the repository belonging to this thesis [Res].

5.11.1 Libraries Comparisons

We will now carry out an evaluation and consideration for each library:

- Annoy: this library has only been tested in section 5.4, section 5.5 and section 5.6 since not distributed and, even more importantly, does not allow a training phase on portions of the dataset smaller than the entirety. This is the worst flaw of this library which, by its nature, is not suitable for medium/large datasets as it would not be possible to load them completely into RAM. Despite this, we still managed to test it on 16, 32 and 64 GB datasets, the results obtained are summarized in Figure 5.4. We can see how this library is the least efficient in terms of search time and actual precision. With 64 GB of dataset we are already over 100 milliseconds and with an accuracy of 87.5%, making this library the worst when compared with FAISS and SPTAG. The question arises spontaneously, does using Annoy make sense? The answer is actually affirmative as the index used is one of those that requires the least tuning phase as explained in chapter 4. We can therefore conclude that this library is an excellent tool for all use cases that need to be performed on relatively small datasets and that do not have time to dedicate to studying the index.
- FAISS: one of the best libraries from every point of view. We mainly tested three types of indices: HNSW, Inverted Index (IVF), Inverted Multi-Index (IMI). The first one has certainly suffered the most from the increasing size of the dataset, as can be seen in Figure 5.4, from every point of view. Despite these imperfections, however, the results remain excellent with a parameter tuning phase that is minimal. The biggest problem with this index, however, is that it cannot be trained on a portion of the dataset. This is due to its internal graph structure. The second and third indices, however, were also tested by varying the size of the training phase, as reported in Figure 5.1. We can see how these two indices are practically interchangeable as they offer almost the same performance in every test case. Despite this, we can see how in the more extreme tests, i.e. those reported in section 5.6 and section 5.7, a small difference in terms of accuracy begins to appear. This difference, although minimal, could increase as the dataset increases as is also communicated in the FAISS documentation [Faig], which reports "*IVF is better for high accuracy regimes, and*

IMI for lower regimes". This information led us to exclude IMI in section 5.8 where we only tested the indexes most used to index such large datasets and with such a negligible training phase [Faif]. We can, in fact, notice excellent results from IVF in Figure 5.2 which allowed us, through a trade-off brought by the parameters, to obtain a precision higher than 60% and search times lower than 100 ms regardless of the other components attached to the index. We can therefore conclude that FAISS provides indexes and solutions that can be adapted to every need with truly competitive index creation and search times and maintaining excellent accuracy even in the most difficult tests.

- SPTAG: this library stands out for being certainly the slowest in the index creation phase. As we can see in Figure 5.4 both indexes tested are extremely slow, taking several hours. This definitely makes this library difficult to use for any use case that needs a quick solution. This very long wait, however, is justified by a very well-balanced data structure as it allows extremely fast searches. We can see in Figure 5.4b that, despite an increase in the dataset, SPTAG manages to maintain search times below 10 milliseconds. Regarding the precision obtained, however, we must distinguish the two indices tested. BKT turns out to be the best index as it manages to maintain extremely high precision even as the dataset increases. This, however, is not true for SPANN which on 64 GB is even less precise than Annoy, which we do not consider suitable for executing on large datasets. Furthermore, thanks to the property of this library to perform training phases on portions of the dataset, we were able to carry out additional tests on these indices, reported in section 5.7. The results obtained, available in Figure 5.1, show us how, however, both indices are unable to be precise if the training portions move away from the totality of the dataset. This leads us to conclude that SPTAG, in particular BKT, is an excellent library if we want to run on large datasets that must be used almost completely in the training phase without performing excessive parameter tuning, leaving this task to the library refining phases as explained in chapter 4, remembering, however, to prepare for long index creation times.

- Distributed-FAISS: this library, being distributed, was tested only in section 5.9 and in section 5.10 on a cluster made up of 8 machines reported in Table 5.3, the results of these tests are available in Figure 5.5. By dividing the dataset equally for each machine we can see how the index creation time is certainly longer than the corresponding creation time if we were to run it on a single machine. This is due to the added delay when transmitting the dataset from the master machine to the nodes on which the index is actually created. The recorded time during the search phase, however, is extremely good despite the delay caused by the transmission as you can see in Figure 5.5b. In fact, we manage to always stay under 50 milliseconds while maintaining precision extremely close to 100%, shown in Figure 5.5c. In our tests we didn't think there was a need to find the perfect parameters to reconcile

search time and precision since even in cases where the precision reaches 100% the search time is less than 50 ms, already considered satisfactory for our cases of study. However, we are certain that it is possible to have a small margin of improvement in search time by sacrificing a couple of percentage points in precision. This can be done through parameter tuning, which is possible so efficiently thanks to our custom implementation [faik]. To be able to affirm the existence of this improvement it is sufficient to refer to the values obtained from the execution of FAISS on a single machine, present in Figure 5.4, since Distributed-FAISS does nothing other than create the same indexes, but on multiple nodes, then performing a search in which it is possible to avoid the k_factor_rf parameter as its behavior is performed by the merge and sort phase by the master node. Among the indexes we tested we have Inverted Index, Inverted Multi-Index and HNSW. In both test cases we trained the indices on the entire dataset and this made them extremely fast and precise. We can conclude that with this dataset and on these machines, the type of index used makes no difference as they are all optimal.

- Distributed-SPTAG: the distributed version of SPTAG has given us a lot of satisfaction and excellent results, as can be seen in Figure 5.5. Unfortunately, however, as already announced in section 5.3, this library only allows the use of the BKT index as the SPANN index is not distributable. We should also note three negative aspects of this library. The first is that it was necessary to write additional code to merge the data returned from each node of the cluster, unfortunately this was not available within the library, as well as to implement a logic relating to the metadata to allow the mapping between the ID of the vector and returned image. The second is that SPTAG's self-trained data structure uses more RAM than the data structures of other indices, this will result in a smaller size of the training portion since the limit imposed by the memory will be reached early. Finally, we must once again point out the times needed to create the index which are disproportionate when compared with other libraries. Despite these negative sides, however, the library has a parameter tuning phase that is practically absent and allows us to obtain truly excellent search precision and speed. We can therefore conclude that, as long as you compromise with the negative aspects, this library is certainly more than valid.

5.11.2 Use Case Comparisons

Finally, trying to find final solutions that can satisfy the majority of use cases, including ours, we would like to suggest, based on the experiments and results obtained, the following approach:

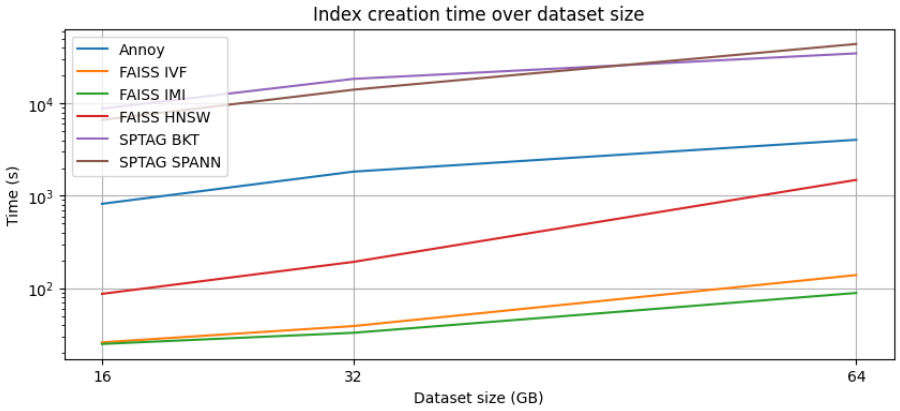
- Small dataset: here the best solutions are Annoy and FAISS with the HNSW index. Both provide excellent performance although Annoy is simpler to integrate into the

codebase, while FAISS with HNSW is slightly more complex but more efficient from all points of view.

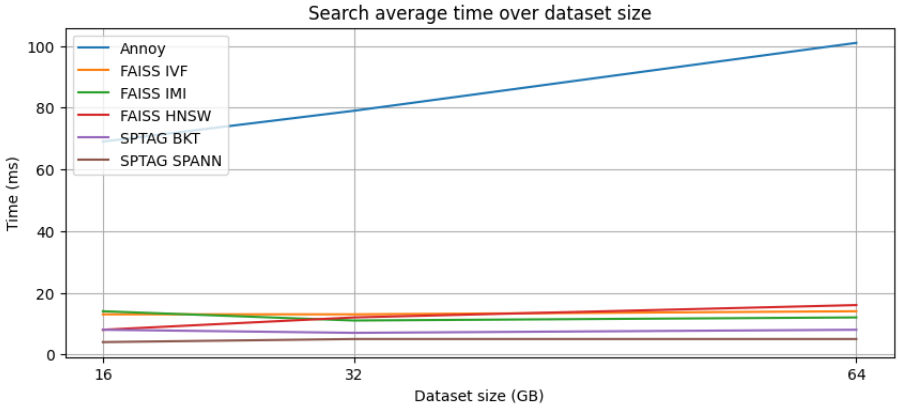
- Medium dataset fully loaded in RAM: in this case, where the entire dataset can be used in the training phase, we recommend SPTAG with the BKT index or FAISS with IVF and IMI. SPTAG certainly provides us with a solution in which we do not have to spend time tuning parameters, but very complex to integrate due to poor documentation. On the other hand, FAISS provides extremely efficient, precise and well-documented indices, but we must start carrying out a parameters tuning phase.

- Medium dataset not fully loaded in RAM: from this moment on we will have datasets large enough that they cannot be fully loaded into RAM on which we will have to carry out training phases on portions of them. In this case the best solution is represented by FAISS with the IVF and IMI index, paying attention to the tuning phase. These solutions are interchangeable as there are no substantial differences between the two.

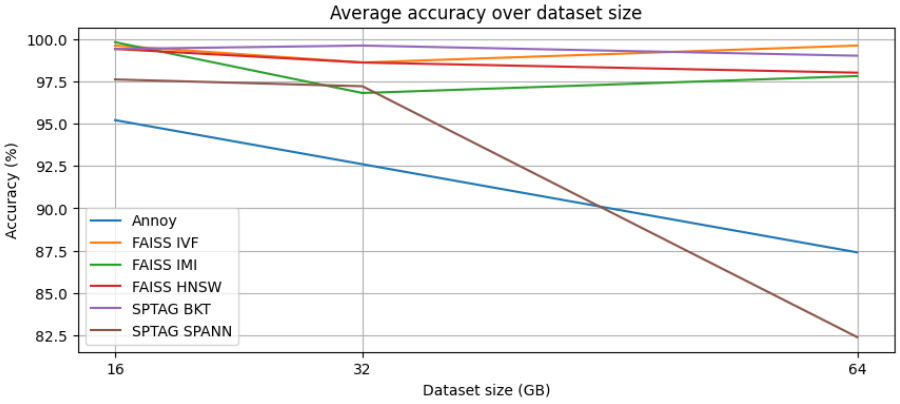
- Big dataset: this use case, corresponding to the one presented in chapter 1 and considered by us to be the most interesting for the purposes of this thesis, involves the use of FAISS and Distributed-FAISS. These solutions are the best and provide excellent results. We recommend the use of the single machine library with Inverted Index, making sure to be able to carry out a training phase on at least 25% of the dataset. If you have a cluster and this is possible on all the machines present in it, then we continue to suggest FAISS in a non-distributed manner to be able to have multiple machines containing the entire dataset and to be able to use a load balancing policy to avoid overloading in case of heavy traffic. If this is not possible then we suggest considering Distributed-FAISS. Even in this last solution we suggest the use of IVF indices trained on the largest possible portion of the dataset to be efficient and precise even on extremely large datasets. However, we would like to remind that the current Distributed-FAISS solution does not allow you to obtain extremely precise results as it requires further implementations, such as the one we proposed and used, available as a pull request on the official FAISS repository [faik].



(a) Creation time changes versus dataset size

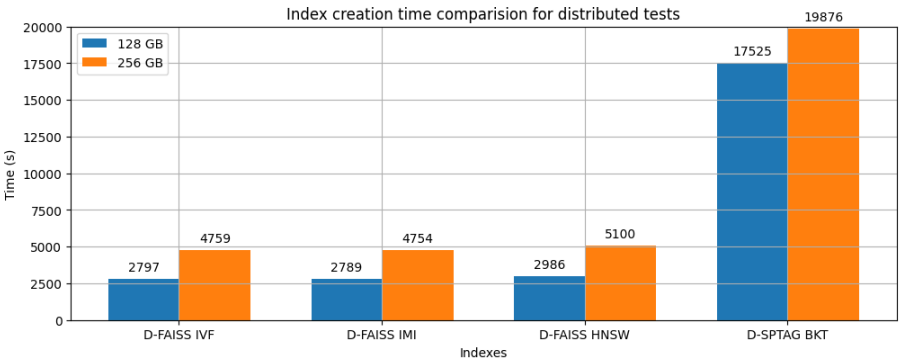


(b) Search time changes versus dataset size

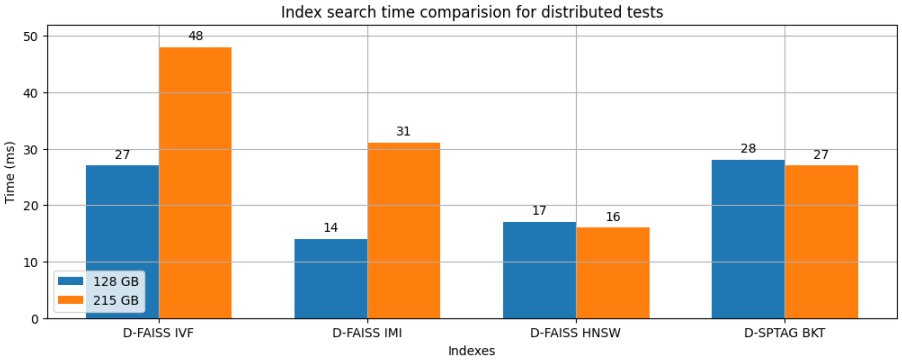


(c) Accuracy changes versus dataset size

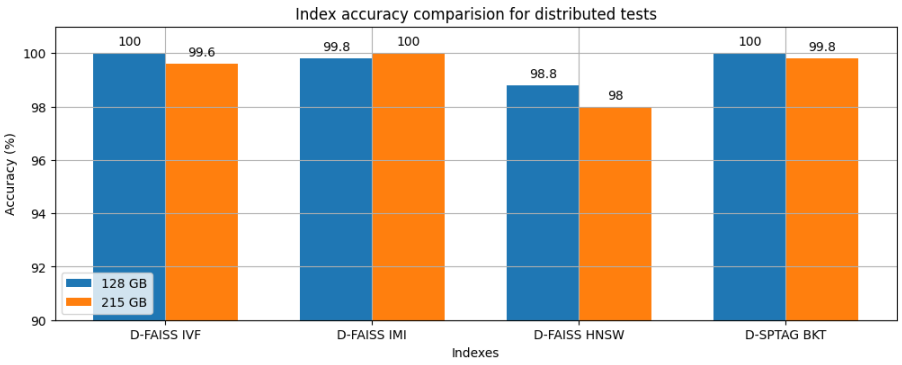
Figure 5.4: Comparison of single powerful machine solutions trained on almost the entire dataset. Graphical display of the data reported in Table 5.6, Table 5.7 and Table 5.8



(a) Creation time changes versus dataset size for distributed tests



(b) Search time changes versus dataset size for distributed tests



(c) Accuracy changes versus dataset size for distributed tests

Figure 5.5: Comparison of distributed machine solutions. Graphical display of the data reported in Table 5.11, and Table 5.12

Chapter 6

Conclusion

In this thesis we presented a family of algorithms called *Approximate Nearest Neighbors* by analyzing the solutions proposed in the scientific literature and their implementations.

Initially, we described the functioning of algorithms capable of executing on a single machine in chapter 2, which provided us with excellent solutions, but which are not able to scale efficiently as the dataset grows. Here we needed to present, in chapter 3, distributed versions of the aforementioned libraries as well as more complex systems, defined vector databases. We subsequently integrated, in chapter 4, the functionality of all the libraries within a codebase [the] that allowed us to carry out tests on the dataset at our disposal, thus discussing the available implementations, the maturity of the libraries and the documentation provided. Finally, in chapter 5, we actually ran the code.

We initially compared the executions on a single machine by training the indices on almost the entire dataset. By repeating these tests as the dataset increases, we realized how and when the accuracy decreases and execution times increase. We subsequently tested the libraries by keeping the size of the dataset constant, but decreasing the percentage of it used in the training phase. In this way, we were able to filter out the indexes that were not suitable for this test case and compare those that actually were. Once this was done we wanted to show how the indexes performing the best from the previous phases behaved on the entire dataset at our disposal by specifically executing them on a machine with limited resources to show how even with a small amount of RAM it is possible to index enormous quantities of data with excellent precision and excellent execution times. After the single-machine tests, we also presented the distributed ones, performed on a cluster of eight machines. In this way, we were able to show how the libraries and indexes behave under both scaling approaches: vertical scaling and horizontal scaling.

We finally collected all the results and, by comparing them, we were able to understand which is the best use case for each library by taking into consideration the strengths and

weaknesses of each. At the end of this comparison, we wanted to conclude with what we set ourselves at the beginning of this document, that is, a guide that could indicate to each user or company which is the best library, the best index and the best approach to follow for their use cases.

We consider have therefore managed to contribute to the scientific literature with a document capable of providing production-ready answers to the majority of use cases in which this family of algorithms is applicable.

Acknowledgements

In this section I would like to thank those who guided me in drafting this document and contributed to improving its quality. Firstly, I would like to mention my supervisor, Professor *Matteo Dell'Amico*, for the excellent support provided and the valuable advice given. Particular thanks then go to the external supervisor, Dr. *Miriam Redi*, and to *Wikimedia Foundation*, who made this work possible by providing me with the use case and the datasets on which it is based. I also would like to express my gratitude to my reviewer, Professor *Daniele D'Agostino*, for the technical support provided in the experimental phases and for the given advice. Finally, I want to thank Professor *Marina Ribaudo* for her availability and the kindness she has shown towards me in these years.

The other acknowledgments will be dedicated to people not directly involved in the technical domain and, therefore, will be expressed in Italian.

Ci tengo ad iniziare questi ringraziamenti nominando le persone che mi sono state più vicine in tutti questi anni, che mi hanno cresciuto, ispirato e che mi hanno aiutato a diventare la persona che sono. Grazie *Mamma* e *Papà*, sono estremamente orgolioso di avere voi come genitori tanto quanto lo sono di dedicarvi questo mio lavoro e traguardo. Voglio, inoltre, ringraziare i miei *zii*, *cugini* e soprattutto i miei *nonni*. La vostra costante presenza nelle varie fasi della mia vita e la vostra fiducia in me sono state un pilastro fondamentale.

Ci tengo anche a ringraziare una persona che è arrivata nella mia vita da poco più di un anno, ma che mi è sempre stata accanto, che mi ha supportato nei momenti difficili e che mi ha sempre fatto sentire importante. Averti al mio fianco rende felice ogni mia giornata. Grazie *Samira*.

Desidero, inoltre, esprimere la mia riconoscenza a tutti i miei amici e compagni di vita, che hanno reso il mio cammino accademico e personale una straordinaria esperienza colma di calore e affetto. Un grazie speciale va ad *Ale*, *Cimi*, *Goss*, *Porci* e *Raffo* (il miglior coinquilino) per le meravigliose giornate, avventure e serate trascorse insieme. Desidero ringraziare *Anna* per le innumerevoli chiacchierate, confidenze e preziosi confronti, e *Samia* per avermi ospitato e fatto sentire fin da subito un amico di lunga data. Non posso dimenticare gli amici di Albisola, una piccola oasi di felicità, in particolare *Sofia* e *Giulia*, per i bei momenti, il sostegno e il legame speciale che si è creato nel corso di questi anni.

Infine vorrei ringraziare i miei *colleghi* di *Blue Reply* per avermi messo nelle migliori condizioni per conciliare studio e lavoro oltre ad avermi sostenuto nella mia crescita personale e professionale.

List of Figures

List of Tables

Bibliography

[20119] Proceedings - 2018 ieee international conference on big data, big data 2018, 2019.

[Ama] Amazons3: https://aws.amazon.com/it/s3/. https://aws.amazon.com/it/s3/.

[anna] ann-benchmarks website: https://ann-benchmarks.com/. https://ann-benchmarks.com/.

[Annb] Approximate nearest neighbor methods and vector models – nyc ml meetup. https://www.slideshare.net/erikbern/approximate-nearest-neighbor-methods-and-vector-models-nyc-ml-meetup.

[annc] erikbern/ann-benchmarks: https://github.com/erikbern/ann-benchmarks. https://github.com/erikbern/ann-benchmarks.

[Annd] Nearest neighbor methods and vector models – part 1: https://erikbern.com/2015/09/24/nearest-neighbor-methods-vector-models-part-1. https://erikbern.com/2015/09/24/nearest-neighbor-methods-vector-models-part-1.

[Anne] Nearest neighbors and vector models – part 2 – algorithms and data structures: https://erikbern.com/2015/10/01/nearest-neighbors-and-vector-models-part-2-how-to-search-in-high-dimensional-spaces.html. https://erikbern.com/2015/10/01/nearest-neighbors-and-vector-models-part-2-how-to-search-in-high-dimensional-spaces.html.

[Annf] spotify/annoy: https://github.com/spotify/annoy. https://github.com/spotify/annoy.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18, 1975. doi:10.1145/361002.361007.

[BL15] Artem Babenko and Victor Lempitsky. The inverted multi-index. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37, 2015. doi:10.1109/TPAMI.2014.2361319.

[Cla94] Kenneth L. Clarkson. Algorithm for approximate closest-point queries. 1994. doi:10.1145/177424.177609.

[CRSW22] Diego Cifuentes, Kristian Ranestad, Bernd Sturmfels, and Madeleine Weinstein. Voronoi cells of varieties. *Journal of Symbolic Computation*, 109, 2022. doi:10.1016/j.jsc.2020.07.009.

[CWL+18] Qi Chen, Haidong Wang, Mingqin Li, Gang Ren, Scarlett Li, Jeffery Zhu, Jason Li, Chuanjie Liu, Lintao Zhang, and Jingdong Wang. *SPTAG: A library for fast approximate nearest neighbor search*, 2018. URL: https://github.com/Microsoft/SPTAG.

[CZW+21] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. Spann: Highly-efficient billion-scale approximate nearest neighbor search. In *35th Conference on Neural Information Processing Systems (NeurIPS 2021)*, 2021.

[data] Wikimedia foundation dataset with images: https://analytics.wikimedia.org/published/datasets/one-off/caption_competition/training/image_pixels/. https://analytics.wikimedia.org/published/datasets/one-off/caption_competition/training/image_pixels/.

[datb] Wikimedia foundation dataset with vectors: https://analytics.wikimedia.org/published/datasets/one-off/caption_competition/training/resnet_embeddings/. https://analytics.wikimedia.org/published/datasets/one-off/caption_competition/training/resnet_embeddings/.

[DCL11] Wei Dong, Moses Charikar, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. 2011. doi:10.1145/1963405.1963487.

[DF08] Sanjoy Dasgupta and Yoav Freund. Random projection trees and low dimensional manifolds. 2008. doi:10.1145/1374376.1374452.

[DFa] facebookresearch/distributed-faiss: https://github.com/facebookresearch/distributed-faiss. https://github.com/facebookresearch/distributed-faiss.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51, 2008. doi:10.1145/1327452.1327492.

[DIIM04] Mayur Datar, Piotr Indyk, Nicole Immorlica, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. 2004. doi:10.1145/997817.997857.

[Dis] microsoft/diskann: https://github.com/microsoft/DiskANN. https://github.com/microsoft/DiskANN.

[DS14] György Dósa and Jiří Sgall. Optimal analysis of best fit bin packing. volume 8572 LNCS, 2014. doi:10.1007/978-3-662-43948-7_36.

[Faia] facebookresearch/faiss: https://github.com/facebookresearch/faiss. https://github.com/facebookresearch/faiss.

[Faib] Faiss github wiki for composite indexes https://github.com/facebookresearch/faiss/wiki/Faiss-indexes-(composite).

[Faic] Faiss github wiki for index factory https://github.com/facebookresearch/faiss/wiki/The-index-factory.

[Faid] Faiss github wiki, indexes section https://github.com/facebookresearch/faiss/wiki/#faiss-indexes.

[Faie] Faiss github wiki search parameters https://github.com/facebookresearch/faiss/wiki/Index-IO,-cloning-and-hyper-parameter-tuning.

[Faif] Faiss github wiki to address 1g vectors https://github.com/facebookresearch/faiss/wiki/Indexing-1G-vectors.

[Faig] Faiss github wiki to address 1m vectors https://github.com/facebookresearch/faiss/wiki/Indexing-1M-vectors.

[Faih] Faiss github wiki to address 1t vectors https://github.com/facebookresearch/faiss/wiki/Indexing-1T-vectors.

[Faii] Faiss github wiki to choose an index https://github.com/facebookresearch/faiss/wiki/Guidelines-to-choose-an-index.

[faij] Faiss wiki: urlhttps://github.com/facebookresearch/faiss/wiki.

[faik] Set index parameter - distributed faiss pull request.

[FXWC18] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. volume 12, 2018. doi:10.14778/3303753.3303754.

[GE89] Jacob E. Goodman and Herbert Edelsbrunner. Algorithms in combinatorial geometry. *The American Mathematical Monthly*, 96, 1989. doi:10.2307/2325168.

[GHKS13] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized product quantization for approximate nearest neighbor search. 2013. doi:10.1109/CVPR.2013.379.

[GLX+22] Rentong Guo, Xiaofan Luan, Long Xiang, Xiao Yan, Xiaomeng Yi, Jigao Luo, Qianya Cheng, Weizhi Xu, Jiarui Luo, Frank Liu, et al. Manu: a cloud native vector database management system. *Proceedings of the VLDB Endowment*, 15(12):3548–3561, 2022.

[Gra84] R. Gray. Vector quantization. *IEEE ASSP Magazine*, 1(2):4–29, 1984. doi:10.1109/MASSP.1984.1162229.

[HAYSZ11] Kiana Hajebi, Yasin Abbasi-Yadkori, Hossein Shahbazi, and Hong Zhang. Fast approximate nearest-neighbor search with k-nearest neighbor graph. 2011. doi:10.5591/978-1-57735-516-8/IJCAI11-222.

[HKJR19] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. 2019.

[hns] Hnsw towardsdatascience article https://towardsdatascience.com/similarity-search-part-4-hierarchical-navigable-small-world-hnsw-2aad4fe87d37.

[IM98] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. 1998.

[Iwa16] Masajiro Iwasaki. Pruned bi-directed k-nearest neighbor graph for proximity search. volume 9939 LNCS, 2016. doi:10.1007/978-3-319-46759-7_2.

[JDS11] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33, 2011. doi:10.1109/TPAMI.2010.57.

[JKG08] Prateek Jain, Brian Kulis, and Kristen Grauman. Fast image search for learned metrics. 2008. doi:10.1109/CVPR.2008.4587841.

[Kle00a] Jon Kleinberg. The small-world phenomenon: An algorithmic perspective. 2000. doi:10.1145/335305.335325.

[Kle00b] Jon M. Kleinberg. Navigation in a small world. *Nature*, 406, 2000. doi:10.1038/35022643.

[KLL+97] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. 1997.

[kub] kubernetes: https://kubernetes.io/.

[LC20] Chen Luo and Michael J. Carey. Lsm-based storage techniques: a survey. volume 29, 2020. doi:10.1007/s00778-019-00555-y.

[LMGY05] Ting Liu, Andrew W. Moore, Alexander Gray, and Ke Yang. An investigation of practical approximate nearest neighbor algorithms. 2005.

[LZS+20] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. Approximate nearest neighbor search on high dimensional data - experiments, analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering*, 32, 2020. doi:10.1109/TKDE.2019.2909204.

[MHL+92] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17, 1992. doi:10.1145/128765.128770.

[Mila] github/milvus: https://github.com/milvus-io/milvus. https://github.com/milvus-io/milvus.

[Milb] Milvus: Billionth-scale similarity search in milliseconds (https://www.youtube.com/watch?v=V5L0DuAMlVc). https://www.youtube.com/watch?v=V5L0DuAMlVc.

[milc] milvus documentation: https://milvus.io/docs.

[ML14] Marius Muja and David G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36, 2014. doi:10.1109/TPAMI.2014.2321376.

[MP69] Marvin Minsky and Seymour Papert. Perceptrons: expanded edition. *MIT Press Cambridge MA*, 522, 1969.

[MPLK14] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45:61–68, 2014. URL: https://www.sciencedirect.com/science/article/pii/S0306437913001300, doi:https://doi.org/10.1016/j.is.2013.10.006.

[MUS18] Yusuke Matsui, Yusuke Uchida, and Shin'ichi Satoh. A survey of product quantization, 2018. doi:10.3169/mta.6.2.

[MY18] Yu A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42, 2018. doi:10.1109/TPAMI.2018.2889473.

[NS06] David Nistér and Henrik Stewénius. Scalable recognition with a vocabulary tree. volume 2, 2006. doi:10.1109/CVPR.2006.264.

[PPK+21] Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Dmytro Okhonko, Samuel Broscheit, Gautier Izacard, Patrick Lewis, Barlas Oguz, Edouard Grave, Wen-tau Yih, and Sebastian Riedel. The web is your oyster - knowledge-intensive NLP against a very large web corpus. *CoRR*, abs/2112.09924, 2021. URL: https://arxiv.org/abs/2112.09924, arXiv:2112.09924.

[Res] Experimental results obtained: https://github.com/ChriStingo/Distributed-computation-of-Approximate-Nearest-Neighbors/tree/main/Docs.

[skl] scikit-learn knn: https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html.

[SKRC10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. 2010. doi:10.1109/MSST.2010.5496972.

[slu] slurm: https://slurm.schedmd.com/.

[sub] Submitit: https://github.com/facebookincubator/submitit.

[the] Distributed computation of approximate nearest neighbors: https://github.com/ChriStingo/Distributed-computation-of-Approximate-Nearest-Neighbors. https://github.com/ChriStingo/Distributed-computation-of-Approximate-Nearest-Neighbors.

[Tou80] Godfried T. Toussaint. The relative neighbourhood graph of a finite planar set. *Pattern Recognition*, 12, 1980. doi:10.1016/0031-3203(80)90066-7.

[wika] Wikimedia foundation: https://wikimediafoundation.org/. https://wikimediafoundation.org/.

[wikb] Wikipedia: https://en.wikipedia.org/wiki/Main_Page. https://en.wikipedia.org/wiki/Main_Page.

[WL12] Jingdong Wang and Shipeng Li. Query-driven iterated neighborhood graph search for large scale indexing. 2012. doi:10.1145/2393347.2393378.

[WTF09] Yair Weiss, Antonio Torralba, and Rob Fergus. Spectral hashing. 2009.

[WWJ+14] Jingdong Wang, Naiyan Wang, You Jia, Jian Li, Gang Zeng, Hongbin Zha, and Xian Sheng Hua. Trinary-projection trees for approximate nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36, 2014. doi:10.1109/TPAMI.2013.125.

[WWZ+12] Jing Wang, Jingdong Wang, Gang Zeng, Zhuowen Tu, Rui Gan, and Shipeng Li. Scalable k-nn graph construction for visual descriptors. 2012. doi:10.1109/CVPR.2012.6247790.

[WYG+21] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2614–2627, 2021.

[WZ19] Jingdong Wang and Ting Zhang. Composite quantization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 41, 2019. doi:10.1109/TPAMI.2018.2835468.

[WZS+18] Jingdong Wang, Ting Zhang, Jingkuan Song, Nicu Sebe, and Heng Tao Shen. A survey on learning to hash. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40, 2018. doi:10.1109/TPAMI.2017.2699960.

[XWL+11] Hao Xu, Jingdong Wang, Zhu Li, Gang Zeng, Shipeng Li, and Nenghai Yu. Complementary hashing for approximate nearest neighbor search. 2011. doi:10.1109/ICCV.2011.6126424.

[ZDW14] Ting Zhang, Chao Du, and Jingdong Wang. Composite quantization for approximate nearest neighbor search. volume 3, 2014.